UNIVERSITETET I OSLO
Institutt for informatikk

Evolvable Hardware Applied to Hand Prosthesis Control

Hovedoppgave

Vidar Engh Skaugen

19. august 2004

Abstract

This thesis is about the use of Evolutionary Algorithms to design a better prosthetic hand controller. One of the goals is to use methods that are easy to implement in a small, low-power and low-cost system. The data set used is typical of the data that would be available in a real-world prosthesis. It was collected by Kajitani at the National Institute of Advanced Industrial Science and Technology (AIST) from a person who had lost a hand, and no advanced preprocessing of the signal was done. Evolutionary Algorithms are used to evolve a digital circuit which can predict the intended hand motion from the data presented to it. The data set is then analyzed to determine the factors that limit the successful classification of signals. The maximum classification rate attainable is determined, and the expected maximum real-word performance is also evaluated. Finally, a method is found that improves the average classification rate at the cost of increased response time. Compared to another work using the same data set, the average classification rate for the testing data rose from 55.1% to 71.2%, for the training data it rose from 73.1% to 92.3%.

Preface

This thesis was written as part of a masters degree at the University of Oslo (UiO), Norway. It was written using LyX, which is a graphical user interface (front-end) for LaTeX (which again is based on TeX).

I would like to thank my adviser, Jim Tørresen, for everything. His presentation on Evolvable Hardware during an informal meeting for post-graduates is the reason I first became interested in this field. He has kept my interest in the field high throughout the whole period, among other things by making it possible for me to attend the ICES'03 conference.

He also took me along to visit Cypromed, an orthopedic workshop located in Hamar, Norway. They have 60-70% of the market for hand prosthesis in Norway, and create around 300 new prosthesis a year. I would like to thank Sten Kvanvik, Bjørn Ludvik Lien, and Trond Peder Schonhowd for showing us how prosthesis are manufactured, how the electrical system of prosthesis works, and how prosthesis are fitted to the user.

I would also like to thank my family, and especially my parents, Siri Engh and Erik Skaugen, for their support, patience, and understanding during my work on this thesis.

I programmed the software used for evolution in the thesis myself. I decided to program my own software instead of using one of the many available software libraries/programs available for several reasons. One is that I like to program and have been doing it for 20 years now. By programming everything myself, I have gained more detailed knowledge about the technical parts of the algorithms used in this thesis. And I have also been more easily able to experiment with unusual algorithms since I know the inner workings of the program, and can change data structures and program flow to suit my whims.

Normally program code is appended as an appendix to the thesis. The program in its current version is too big to be included however. An approximate 500 pages would be needed for the whole program. Selecting parts of the program and presenting them is also difficult. Some of the programming methods used make even simple functions impossible to understand without a good understanding of the program as a whole. Some of the methods used are self-modifying code, extensive object caching and reuse, callback functions and thread-synchronization. Also, program flow switches between 5-15 different objects (from a pool of over 100) depending on the configuration file for the current run.

Maybe I'll make a simpler interface and document the program properly someday. At present, the program continues to grow and change as I learn more and more about Evolutionary Algorithms.

- Vidar Engh Skaugen, August, 2004.

Contents

1 Introduction

In the United States of America 41.000 persons are registered as having had an amputation of a hand or a complete arm[8]. This equals 1 in 6100, which if applied on a global scale means around 1.000.000 people worldwide with an amputation of a hand or a complete arm. The loss of an upper limb results in drastic reduction of function and cosmetics. To provide these people with a higher quality of living, a lot of research on prosthetic limbs has been done in the last decades, initially spurred on by the large number of people needing prosthesis after World War II.

There are three main types of prosthesis available: Passive, conventional (or body-powered), and electrically-powered prosthesis. The simplest type is the passive prosthesis. They are usually crafted to duplicate the looks of the lost limb, or cosmetic restoration as it is called. This is the most important consideration for many patients. They may also serve as a simple aid in carrying and balancing.

Conventional prosthesis are powered and controlled by gross body movements. These movements, usually of the shoulder, upper arm, or chest are captured by a harness system and transmitted via a cable to a terminal device (hook or hand). This allows the user to open and close the terminal device, and thus grasp and carry items.

The most complex prosthesis is the electrically-powered prosthesis. It uses a battery and electrical motors to open and close the terminal device. The prosthesis can be controlled by switches, touch sensors, or muscle contractions in the forearm (upper arm for above-elbow amputees). Muscle contractions are measured using electromyography (EMG), and electrically-powered prosthesis controlled by EMG signals are called myoelectric prosthesis.

The different types of prosthesis each have their pros and cons. Silcox et al performed a survey on 44 people who had been fitted with a myoelectric prosthesis at Emory University Affiliated Hospitals, from January 1972 through December 1989[15]. Of the 44 patients, 40 (91 per cent) owned a conventional prosthesis and 9 (20 per cent) owned a cosmetic prosthesis in addition to the myoelectric prosthesis. The reason for having several prosthesis was that different types of prosthesis were used for different occasions. For example, the cosmetic prosthesis was often used for social occasions while a more functional prosthesis was used in work situations.

Of the 40 people who owned both a myoelectric and a conventional prosthesis, 50% rejected the myoelectric prosthesis, while 33% rejected the conventional prosthesis. 23% used the conventional prosthesis exclusively. This shows that a significant number of people found the conventional prosthesis better suited for their requirements than the electrically-powered one. The three top reasons cited for non-use of the

Figure 1: Otto Bock SensorHand. Picture copyright©Otto Bock HealthCare.

electrically-powered prosthesis were weight, speed, and lack of durability.

Many of the disadvantages of electrically-powered prosthesis have been addressed in the years since the survey. Advances in battery, motor, and material technology have allowed the production of prosthetic hands that weigh less than a normal human hand, and higher speed prosthesis have also been developed, such as the Otto Bock SensorHand which has a 300mm per second opening/closing speed[29]. A prototype prosthetic hand using flexible fluidic actuators was able to perform complete flexion and extension of a finger in less than 100 ms, making it possible to open and close the hand with a frequency >5 Hz[2].

Durability is still a problem, especially when taking into consideration the much higher costs of an electrically-powered prosthesis. A conventional below-elbow prosthesis costs $1500-$2000 (hand and wrist,

Australian national indicator of cost converted to $USD), while an electrically-powered below-elbow prosthesis costs $13.500-$22.000 (hand only, $USD) or $15.500-$25.000 (hand and wrist, $USD)[24, 25].

The electrically-powered prosthesis does have some advantages to offset its higher costs. In a conventional prosthesis, the need to securely fasten the harness to allow power transference to the terminal device might cause the harness to feel thigh or constricting. This is not a problem with electrically-powered prosthesis, since they provides their own power. The motors can also provide more power and a higher grip force than what is available through a harness system, and it operates at full power in all positions. Conventional prosthesis have a functional envelope, which is the area in space where the patient can control his or her prosthesis. For many the functional envelope is limited to directly in front of them from waist level to mouth level. Significant control reduction occurs when attempting to operate the prosthesis out to the side, down by the feet, and above the head.

Using an electrically-powered prosthesis allows more functions than a conventional prosthesis. In addition to the open/close hand motion, full control of wrist rotation is currently available. Also, new technology is allowing more equipment to be included in the prosthesis, allowing prosthetic hands that more accurately mimic the range of motions available to the human hand. This is no easy task, as the human hand is an extremely complex piece of machinery. It contains at least 27 bones and around 40 muscles (numerous anatomical variations exists), and has approximately 24 degrees of freedom[13].

Prosthetic hands that have 10 and 18 degrees of freedom (D.O.F.) have been prototyped, but are not commercially available[13, 2]. The higher number of D.O.F. means that the user has to transmit more information to the prosthetic in order to control it. This is a problem, since the systems used to control current commercial hands do not scale up well to additional D.O.F.

1.1 EMG signals

The most common way of controlling prosthetic hands is by using EMG signals. Usually EMG sensors, or electrodes, are placed on the skin above a muscle in the amputated limb. When the muscle contracts, a weak electrical signal can be detected through the skin by the EMG sensors. The location of the muscles that are activated depends on the motion performed. Generally speaking, to open the hand, the muscles located on the **underside/bottom** of the forearm when the hand is facing upwards need to contract. To close the hand, the muscles located on the **top** of the forearm when the hand is facing upwards need to contract. This makes it possible to predict hand motions by evaluating the electrical

signals recorded by the EMG sensors.

The location of the EMG sensors are decided on a case by case basis. Good candidate locations for EMG sensors are on the skin directly above a big muscle, since the EMG signal tends to be strongest there. The user should be able to activate the muscle group without activating other muscle groups used to control the prosthesis. Also, scar tissue and muscle damage can prevent EMG sensors from getting a good signal.

In cases where the EMG signal is too weak, an alternative control system such as a control box with switches or a harness systems can be used. The harness system used to control electrically-powered prosthesis does not directly transfer power to a terminal device like a conventional prosthesis, but can for example measure the tension placed on a strip of fabric and generate an electrical signal used to control the prosthesis. Research is also being done on neural interfaces, which are small devices implanted inside the body that interface directly to the nervous system[7]. Since this is a relatively new and very invasive procedure that has mostly been tested on auimals, it is not currently used in commercial products.

1.2 Control strategies for prosthesis

With most current prosthesis, each EMG sensor is used to control exactly one motion. For example, once the strength of the signal exceeds a set value, the prosthetic hand opens. The strength of the signal can also be used for proportional control. The stronger the signal, the faster the hand opens.

One problem is that EMG signals for a single individual may vary in strength over time. Gaining or losing weight will change the thickness of the layer of fat beneath the skin, which will affect the strength of the EMG signal recorded. Muscle fatigue also affects the EMG signal, causing it to get weaker. Compensating for this adds extra complexity to the prosthesis. Only recently was a prosthesis that was able to auto-calibrate itself to compensate for these changes made commercially available, the *Utah Procontrol 2* hand[27].

If more functions than opening and closing the hand are desired, a more complex control system is required. Additional EMG sensors can be used to get more control signals. One system that uses this method is the *13E195 Otto Bock Four Channel Processor II*, which is used together with a wrist rotator and a prosthetic hand to allow rotation of the wrist in addition to the normal functions of the prosthetic hand[28]. This requires two additional EMG sensors, one to control wrist rotation clockwise, and one to control rotation counterclockwise. The additional sensors and components increase the complexity and cost of the prosthesis

Another alternative is to use one set of sensors and have some signal that switches between the different functions. The *Utah Procontrol 2* hand mentioned above uses two EMG sensors to open and close the hand as normal[27]. A rapid co-contraction of both muscles will switch control to the wrist, allowing wrist rotation using the same two EMG sensors. Another rapid co-contraction switches control back to the hand. Only one function can be used at a time, but only two EMG sensors are needed.

Usually a long training period (almost one month) is required before multi-function myoelectric prosthetic hands can be controlled[4]. However, it is also possible to make the prosthesis and its control system try to adapt to the user, rather than the other way around.

The EMG sensors are placed as normal on a person, and that person is then asked to perform the motions that the prosthetic is to duplicate. For example, a person with a missing hand could be asked to focus on opening his hand. The signals are recorded together with a value indicating which motion the signal was for. This is repeated for each motion the prosthetic is to reproduce, and several samples are taken for each motion. This data set is then analyzed to find good rules for predicting which motion the user intended.

It it hard to make a system that works satisfactory for all persons using this method, since EMG signals vary from person to person. This is especially true for amputees. When surgically amputating a hand above the wrist, the muscles in the forearm are stitched together (or directly to the bone) to form a layer of soft tissue around the end of the bone. In addition to changing the layout of muscles in the forearm, the muscles that were formerly used to control the hand will not be used actively, and may atrophy over time.

This means the control system has to be personalized for each user. The most common way to solve this problem is by using Neural Networks or Evolutionary Algorithms, which will be covered in the next chapter.

1.3 Preprocessing EMG signals

The EMG signal consists of small spikes or impulses that are only a few milliseconds long. These impulses travel along the cell membrane of the muscle fibers and start a chemical reaction that eventually leads to muscle contraction. A single impulse will only lead to a short muscle twitch, for continuous contraction a series of impulses are needed.

It is difficult to work directly on a series of small impulses, therefore the signal is usually preprocessed. This is also called feature-extraction. Two different forms of preprocessing are prevalent, Fourier transforms and integration of the absolute signal. Fourier transforms require specialized hardware or a fast processor. This is a disadvantage when the

Figure 2: Otto Bock Transcarpal Hand. Picture copyright©Otto Bock HealthCare.

control system has to fit within the strict space, weight, and power constraints of a prosthetic limb. Integration is much easier to implement, and is the method used for the data set in this thesis (see Chapter 3).

Other methods, such as logarithm-transformation methods[26] and

μ-law quantization[12], can also be used in addition to Fourier transforms and integration. They have been shown to improve the performance of the control system, but are again dependant on more hardware and are not used here.

1.4 Goals and layout

The goal of this thesis is to use Evolutionary Algorithms to design a better prosthetic hand controller. The research is theoretical, implementing the controller is outside the scope of this thesis. The result should however be as easy as possible to implement in hardware, so the following restrictions are followed:

- It should be possible to implement the result using common, small, inexpensive and low-power components.
- Because of the above, no advnaced functions such as Fourier transforms or mathematical transformations should be used.
- The EMG signals should be typical of those available in a normal prosthetic hand.

Chapter two starts by giving a definition of Evolutionary Algorithms and related terms that are used throughout this thesis. Chapter three gives a description of the data set that is used, and shows the resuls of some other works on posthetic hand control. The initial results of applying Evolutionary Algorithms to the data set are shown in chapter four. In chapter five the data set is analyzed further based on the initial results. Chapter six considers ways of improving the data set, and chapter seven presents the results of the evolutions. In chapter eight the conclusion is presented. Thoughts on future work are shown in chapter nine.

2 Evolutionary Algorithms

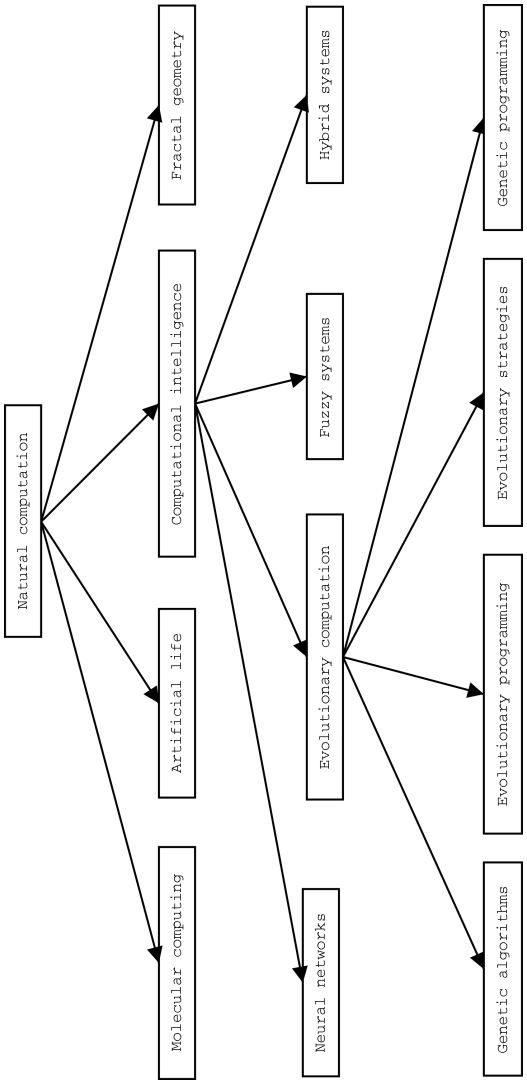


Figure 3: Relationship of fields in Natural Computation.

Most works on prosthetic control use either an Evolutionary Algorithm (EA) or a Neural Network(NN). In this chapter the basics of how EAs works will be presented, and the terms used throughout the rest of the thesis will also be explained. Using information from several sources[10, 11, 17, 21] I made Figure 3, which shows how EA and NN are related to each other and to the rest of the fields in this branch of science.

At the very top is Natural Computation, which is a general term re-

ferring to computing that is inspired by nature. It is the root of this branch of science. For solving the problem of prosthetic hand control, currently only the field of Computational Intelligence is of interest. Molecular Computing seeks to exploit the computational power inherit in biological systems, for example by using DNA molecules and enzymes to implement algorithms. Fractal geometry has been inspired by the patterns found in nature. Artificial life simulates the life of individuals. Computational Intelligence (or Bio-inspired methods) uses principles and methods found in nature to solve problems.

Within Computational Intelligence there are several different ways to approach the goal of solving problems. Fuzzy systems replace the absolute values of normal computer systems with a graded relationship, IE "This is probably true" or "A is quite a bit larger than B." Neural Networks emulate the neurons of a brain, and can be trained to solve a problem by providing feedback to the system, which can then adjust its behavior accordingly. Evolutionary Algorithms (also known as Evolutionary Computation) emulates the way species evolve by starting with several solutions, then discarding those solutions that do not perform well enough and "breeding" new solutions from the ones that survived. Hybrid systems combine one of the fields in Computational Intelligence with other methods such as regression or statistical techniques.

Before we cover the last part of Figure 3, the differences between the methods in Evolutionary Algorithms, we will first take a closer look at how Evolutionary Algorithms work.

2.1 Coding

The way the solution is coded determines the number of solutions that are available. Consider the following coding of the circuit in Figure 4:

The circuit has 4 inputs and 4 outputs, and consists of 4 x 4 logic blocks. Each logic block has two inputs which can be selected from one of four possible inputs, and each logic block can have 1 of 4 different functions (and, or, nand, xor).

Number of bits per logic block = 2 bits per input * 2 + 2 bits for function type = 6 bits.

Total number of bits = 6 bits * 16 logic blocks = 96 bits.

This means there are 2^{96} possible solutions available. The 96-bits long bit string representing the circuit is called the *genotype*, while the circuit itself is called the *phenotype*. These words have been borrowed from biology, where they are used in a similar manner to distinguish between the genes and the creature that grows from those genes.

Even with a small circuit like this, evaluating all possible solutions will take too long to be practical. Some way of searching through the solutions is needed. The set of all possible solutions is called the *search*

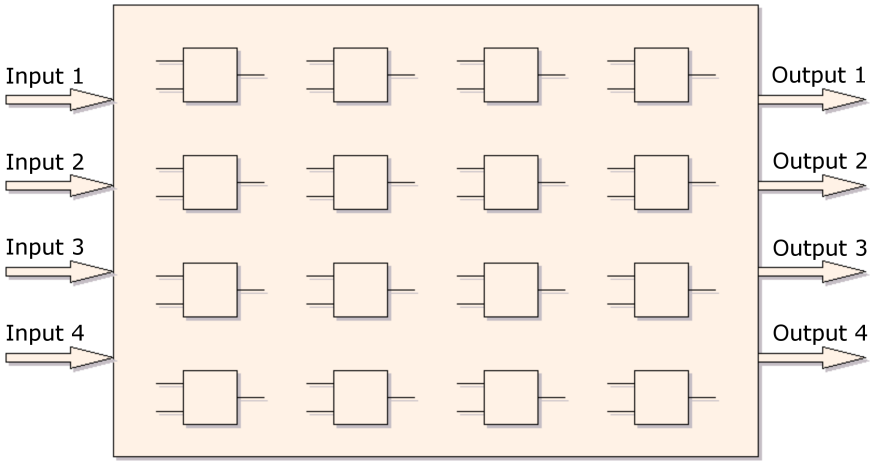


Figure 4: A system consisting of 4x4 logic blocks.

space. To be able to pick the best solution, it is necessary to assign a *fitness value* to each evaluated solution. The function that determines the fitness value is called the *fitness function*.

In this case, there are $2^4 = 16$ possible inputs and $2^4 = 16$ possible outputs. If the goal is to evolve a 2-bit multiplier, the fitness function could compare the inputs and outputs to the truth table for a 2-bit multiplier and count the number of bits that were correct. Each of the 16 possible inputs would produce 4 output bits, and each correct output bit could add 1 to the fitness value. This would give a value between 0 and 64, with a higher value indicating a more fit circuit. Other measures, such as speed and size, could also be integrated into the fitness function. For example, a circuit that utilized fewer than the 16 available logic blocks could be rated higher than a circuit with the same number of correct bits that used all 16 logic blocks.

2.2 Search

The most basic search method available is random search. Random search, as the name implies, randomly picks solutions from the search space and evaluates them. Once a solution is found that is deemed good enough, or a time limit has been exceeded, the algorithms returns the best solution found. If the fitness values are randomly distributed, this method will have the same performance over time as a method that simply evaluates all possible solutions in order.

The fitness values are seldom randomly distributed however. In this case, circuits with only a few bits different will for the most case have similar truth tables. A small change in the genotype usually leads to a

small change in the phenotype. This knowledge of the problem domain can be exploited to give more efficient search mechanisms.

A requirement for this exploitation is a good fitness function. In a worst-case scenario, where the fitness function returns 1 for a perfect solution and 0 for an imperfect solution, the best an algorithm can do is to search randomly until it stumbles across the correct solution. No matter how small the improvement of a solution, there should be a corresponding increase in the calculated fitness value. This allows the population to gradually converge on a good solution, in tiny steps if necessary.

A simple algorithm that utilizes knowledge of the problem domain is the hill climbing algorithm. It picks and evaluates a random solution, then evaluates all adjacent solutions. It then moves on to the solution that gives the largest increase in fitness. In the case of the example with 16 logic blocks above, this might be achieved by flipping the first of the 96 bits, evaluating the created solution, then flipping the bit back to its original value and moving on to the next bit. Once all adjacent solutions have been evaluated in this manner, the process is repeated with the best solution until no better solution can be found.

The problem with this algorithm is that it will stop at local optima. Unless it is lucky enough to start close to the global optima, it will not find the best solution. Combining hill climbing with random search might yield better results. For a wide range of problems, this might be a good solution. There is however another feature of the problem domain that can be exploited, the concept of *building blocks*.

If we consider a solution to be built up of a smaller number of building blocks, it becomes possible to search for those building blocks that tend to give good solutions. By discarding those building blocks that have no positive impact on the fitness, we shrink the search space drastically. If we in our example above could pick out 20 bits that when set to 1 gave no positive impact on the fitness value, we could lock those bits to 0 and reduce the search space by 2^{20}, or roughly one million times. This would make the algorithm a million times more efficient than a random search. One of the features of Evolutionary Algorithms is that they can use the concept of building blocks to search through the areas of the search space most likely to yield good results.

2.3 Genetic Algorithms

The most widely used form of Evolutionary Algorithm, the *Genetic Algorithm*(GA), uses the concept of building blocks. Genetic Algorithms do not explicitly work with the building blocks themselves, but work on a population of individuals to achieve the same effect. It works as follows:

```
Initialize population with random values
REPEAT
  Evaluate all individuals
  REPEAT
    Select two individuals for reproduction
    Use crossover to generate two new offspring
    Mutate offspring
    Place offspring into a new population
  UNTIL new population is full
  Replace population with new population
UNTIL a satisfactory solution is generated
```

The initial population will probably not contain any really good solutions, since all the individuals are initialized with random values. But some of the individuals will be a bit better than the rest, and these have the largest chance of being selected for reproduction. Since the individual is above average, this means that the building blocks that make up that individual are above average as well. Once two individuals are selected, crossover is performed on them. Some building blocks tend to get disrupted during crossover, and are therefore likely to produce offspring with a lower fitness. These offspring will have a lower chance of being selected in the next round of selection, so easily-disrupted building blocks will tend to disappear, even if they add to the fitness of a circuit.

The building blocks that will end up dominating the population are those that add to the fitness of a circuit and that have the best chance of surviving the crossover operation. This limits the search space not only to those solutions that give a good fitness value, but also to those solutions that tend to give good offspring when crossover is performed on them.

2.3.1 Crossover

Crossover is the most important feature for GAs, and need some explanation. In the following example, the solution is coded by using 8 bits. If we write the content of two individuals as AAAAAAAA and BBBBBBBB, crossover works as follows: First a random point between two bits is chosen, the *crossover location*. There are 7 possible crossover locations, in this example no 5 is selected.

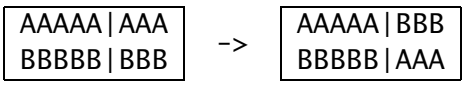


Figure 5: Crossover

The bits after the crossover location are swapped, and the result is two new individuals. This is called *one-point crossover*. Two-point crossover is also possible. Two crossover locations are chosen, and the bits between the two crossover locations are swapped. Look at the following example, where location 1 and 7 is selected:

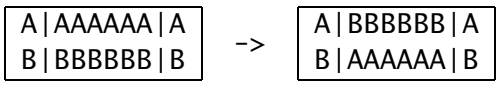


Figure 6: Two-point crossover

No matter what combination of crossover locations are selected, the first and last bit act as a combined building block, they can never be split by the crossover operation. This is easily corrected by adding the location **after** the last bit as a possible crossover location. To illustrate this, look at the following figure, where location 3 and 8 are selected from the 8 possible crossover locations:

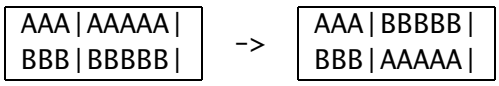


Figure 7: Two-point crossover

This will give the two-point crossover operation the ability to act identically to the one-point crossover operation, and the first and last bits are independent building blocks again.

The crossover operation will tend to disrupt long building block. A building block consisting of 6 bits is much more likely to be split than a building block consisting of 2 bits. Short building blocks that contribute to the fitness are much more likely to survive. In fact, an exponentially increasing number of tries are allocated to short and good building blocks[10].

2.3.2 Mutation

Mutation is used to let as many building blocks as possible get evaluated. Those building blocks that were not present in the initial population, or were lost during evolution, have a chance of being introduced into the population. This is a secondary operator, and the chance of mutation is usually set to a low number. Mutation works by randomly changing a single individual.

Original bit string	011000101101
Select a random bit	01100<u>0</u>101101
Mutate to new bit string	011001101101

Figure 8: Mutation

2.3.3 Selection pressure

When two individuals are selected to generate offspring, individuals with high fitness are favored over those with low fitness. This will indirectly favor the building blocks that contribute to good fitness, and weed out the building blocks that have little or no positive effect. If only a few of the best individuals are allowed to reproduce, the selection pressure is said to be high. If the best individuals are only given a slight advantage over low-fitness individuals, the selection pressure is said to be low.

If selection pressure is too high, all but the most fit building blocks will quickly be weeded out, and since there are no new building blocks to experiment with, the population stagnates. Once the diversity of the population is lost, only the mutation operation will generate new solutions, and most of the benefits of using EAs disappear. The selection pressure has to be high enough so that the population will converge on a good solution within a reasonable timeframe, but not so high that the population will stagnate early in the run.

Roulette selection is a normal method for selecting individuals. Each individual is assigned a probability of being selected that is directly proportional to it's fitness. This can cause above-average individuals to dominate the population early on, so fitness scaling is often used. Fitness scaling scales the fitness values of all individuals so that the best individuals get a good, but not too excessive, benefit over less fit individuals. This prevents stagnation early in the run, and increases selection pressure at the end of a run. Rank-based selection and tournament selection are other alternatives to roulette selection.

2.3.4 Cloning

Cloning is used to copy an individual into the new population without performing mutation or crossover on it. Elitism is when the best individual is always copied into the new population. Use of this method means there is little or no chance of a good solution being lost once it has been found. On the other hand, it lowers the pressure on generating solutions that are good at surviving the crossover operation. Whether or not to use cloning or elitism depends on the problem at hand.

2.4 Types of Evolutionary Algorithms

Genetic Programming is method similar to Genetic Algorithms, except that it uses a tree-based or a variable-length gene representation. It evolves programs from a set of terminal and operand symbols, and uses reproduction and modified versions of the mutation and crossover operations.

Evolutionary Strategies are another type of EA where each individual is represented as a real-valued vector. It does not use crossover, but instead uses a set of control parameters that are mutated together with the normal parameters. For example, the standard deviation to use when mutating values could be included as a control parameter in each individual. Once a good value for the standard deviation has been found, it will tend to produce better offspring. Those individuals without a good values for standard deviation will be at a disadvantage, and will be weeded out of the population. This leads to a more efficient search, since instead of specifying a standard deviation to use throughout the evolution, the algorithm finds the best value to use at any given time during the run. This is called *self-adaptability*. Since the parameters that lead to the most efficient search are dependent on the fitness landscape, Evolutionary Strategies can be said to have their own way of exploiting the fitness landscape. *Evolutionary Programming* uses the same principle of self-adaptability to evolve programs (or transition tables of finite state machines).

The main differences between the four methods within Evolutionary Algorithms can be summed up in Table 1:

Evolutionary Algorithm	Coding	Crossover	Self-adaptation
Genetic Algorithm	Array	Yes	No
Genetic Programming	Program	Yes	No
Evolutionary Strategies	Array	No	Yes
Evolutionary Programming	Program	No	Yes

Coding How the EA codes the solution. Either coded as an fixed-width array, or a program representation (tree or variable-length array).

Crossover Whether the EA uses crossover in addition to mutation or not.

Self-adaptation Whether the algorithm uses self-adaptation or not.

Table 1: Comparison of different EAs.

The algorithms use either crossover or self-adaptation. Both of these

concepts are taken from evolution in nature, and at least one of them has to be present for an algorithm to be considered an Evolutionary Algorithm. The concept of working with a population of individuals is also used to distinguish EAs, but Evolutionary Strategies for example does not need a population to work. One important charasteristic of EAs is that they use probabilistic transition rules, not deterministic rules.

2.5 Evolvable Hardware

Several different definitions for Evolvable Hardware (EHW) exists[18, 16]. EHW is usually used to describe two different things:

1. Designing hardware using Evolutionary Algorithms.
2. Hardware that is able to reconfigure itself to adapt to external events.

A Field Programmable Gate Array (FPGA) programmed to act as a 3-bit multiplier can be classified as EHW, as long as the configuration of the FPGA was evolved. If a human configured the same FPGA by hand to act as a 3-bit multiplier, it would no longer be EHW. The word *hardware* (HW) is a key point in this definition. A typical definition can be found in "The Free On-line Dictionary of Computing"[20]:

Hardware The physical, touchable, material parts of a computer or other system. The term is used to distinguish these fixed parts of a system from the more changeable software or data components which it executes, stores, or carries.

This used to be a very clear definition 40 years ago. There were also several other key points that distinguished software and hardware:

Changeability Only software systems could change their configuration easily. Hardware functions could only be changed by use of a soldering iron.

Size The large number of transistors needed to implement a processor meant that any system capable of running software was large and power-hungry.

Speed CPUs were slow, having low clock speeds and using several clock cycles to perform a single command. Implementing a function in hardware could yield speeds several orders of magnitude faster than a software function.

Function CPUs could be programmed for any function. Hardware implementations typically performed one function only.

But with the introduction of Programmable Logic Devices (PLD) in the beginning of the '70s, the line between software and hardware started to blur. PLDs cannot easily be put into either the hardware or software category, they belong to both in varying degrees. A PLD's function is undefined at the time of manufacture, and is programmed at a later time. The most common types of PLDs today are Field Programmable Gate Arrays (FPGA) and Complex PLDs (CPLD).

In 1971 the world's first commercial microprocessor was released, the 4-bit 4004. Having a CPU on a single chip greatly reduced the size difference between hardware and software systems.

In 1972-73 the first fuse-link one-time Programmable Logic Arrays (PLA) were designed. These were programmed by blowing fuses inside the chip. The chip could only be programmed once, and the programming permanently changed the physical layout of the chip (by breaking electrical connections). The PLA had the speed of hardware and could not be reprogrammed like software, so it was natural that these devices were categorized as hardware.

A PLD that could also be erased using UV light arrived later in the '70s, and in 1985 Xilink introduced FPGAs, which held the configuration in static RAM and could be reconfigured on-board. The earlier PLDs starting off as mostly hardware and gradually took on more and more software aspects, but were still considered hardware.

PLDs have the unique characteristic of being defined as hardware while still being easily changeable. Another type of EHW is slowly becoming feasible, hardware that is able to evolve its physical structure to solve the problem at hand. Nanobots are an example of how EHW in the future might reconfigure its physical structure to solve the problem at hand.

With this new technology becoming available, the following definition might be more precise:

1. Designing a circuit diagram or PLD configuration using Evolutionary Algorithms.

2. Hardware that is able to reconfigure its physical aspects or PLD configuration to adapt to external events.

3 Hand prosthetics data set

Kajitani at the National Institute of Advanced Industrial Science and Technology (AIST) in Japan has been working with Evolvable Hardware (EHW) to create a prosthetic hand control system that can control several functions at once, and at the same time be able to adapt to the user[5, 4, 26, 12]. During this research, a set of EMG signals were recorded from a person who had lost a hand.

There are still very few works to date that have used EMG signals from an amputee. Most recordings are taken from a person without amputations. The benefits of this data set it that it closely mimics the conditions that are typically found in a prosthesis. The user is an amputee, the total number of bits used for the input vector is limited to 16, and no advanced preprocessing methods are used. Four EMG sensors are used. Using fewer sensors would be preferrable, as the prosthetic could be made cheaper and less complex. This is a consideration for future work.

To illustrate the method used when collecting the data set, I made Figure 9. Four EMG sensors (1) were connected to a person missing a hand. The person concentrated on one motion (for example supination of the wrist), and the absolute value from the four EMG sensors were integrated over a period of one second (2). A run of 10 input vectors were generated concurrently, with a delay of 100ms (3) between each input vector. In the illustration, only the first few input vectors are show. The first (2) and second (4) input vectors, and an outline of the third input vector (5). Since sampling of the last input vector starts 900ms later than sampling of the first input vector, samples were continually taken for 1.9 seconds while the person concentrated on a motion.

Each of the four channels were quantized using four bits (6). The values from the four channels were then concatenated to form a 16-bit value (7). The motion the person concentrated on was also stored together with the input vector. 10 runs were done for each motion, so each motion had a total of 100 input vectors. Data for 6 different motions (three different degrees of freedom) was collected: open and close hand, extension and flection of wrist, pronation and supination of wrist.

Of the 600 total input vectors, half was used in a training set and half in a testing set. Training the circuit on all possible input vectors would require an enormous data set and require too much time. Therefore the circuit is only trained on a small subset of all possible input vectors, the ones contained in the training set. The testing set is used to estimate how well the circuit will perform in a real-world situation. The performance for the training set is usually higher than the performance for the testing set. Since the testing set contains values the circuit has

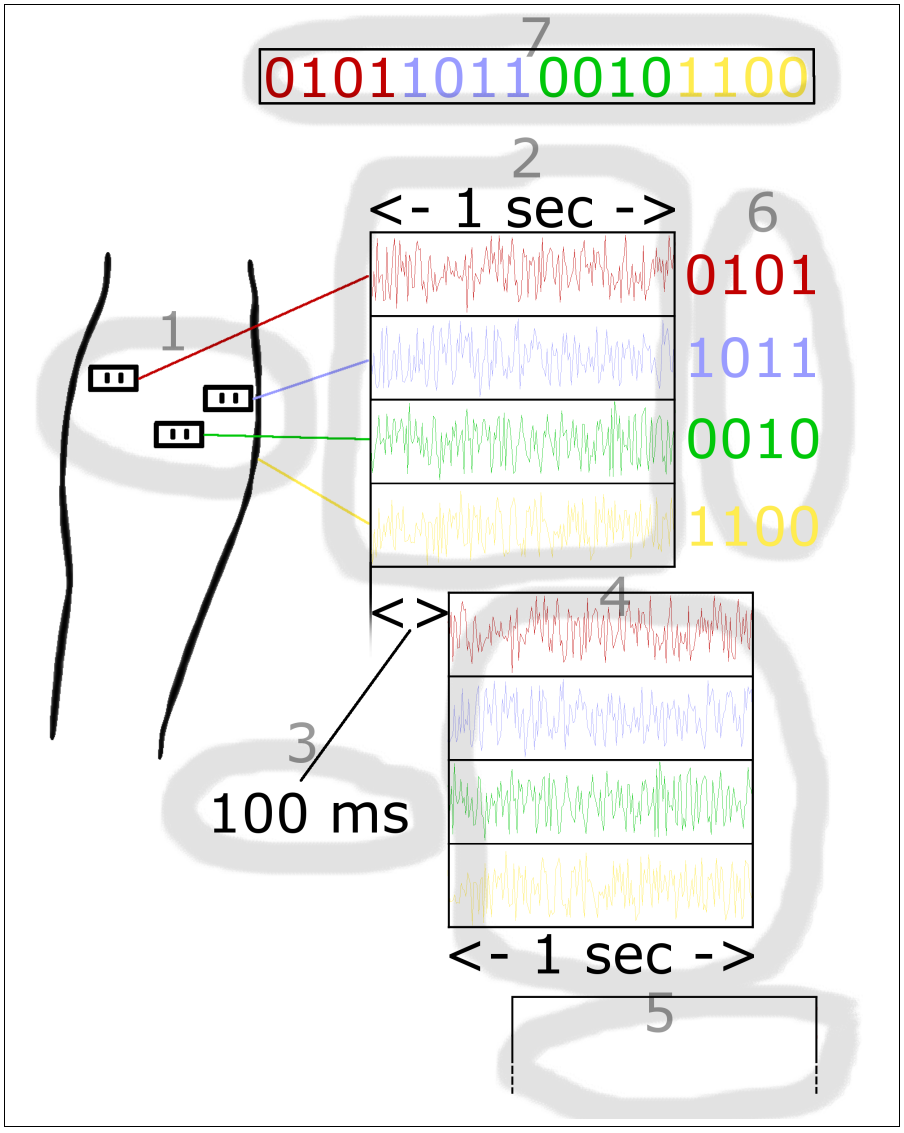


Figure 9: An overview of how the data set was collected.

not encountered during its training, it should give a good indication of real-world performance.

This data set has not been used in any of Kajitani's own work to date. It has been used by Tørresen, who studied with Kajitani during his research work in Japan. Some articles that work with EMG datasets will be presented below, starting with Tørresen's article.

3.1 Results by Tørresen

Using the dataset described above, Tørresen published an article about using two-step incremental evolution to find a circuit that could convert the EMG signals to the intended motion[23].

In the article, Tørresen first evolved six different circuits using an AND-OR layout that each try to predict a single motion. Each circuit has 32 inputs (the 16 bits from the input vector and their complements), a layer of 32 AND gates, and a layer of 32 OR gates. When an input vector that corresponds to the motion the circuit is trying to predict is input, a large number of **high** outputs from the OR layer increases the fitness. When an input vector that **does not** correspond to the motion the circuit is trying to predict is input, a large number of **low** outputs from the OR layer increases the fitness.

A scaling factor, $s = 4$, was used to emphasize input vectors for the motion the circuit was trying to detect. Each correct high output gave four times as much fitness as a correct low output. Also, he used either 16 or 32 output values (referred to as *fitness measure*) to calculate the fitness for the circuit, and the number of inputs to each AND/OR gate could be set to two, three, or four.

In the second stage of the evolution, a selector unit and a counter unit is added to the outputs of each of the six circuits. The selector units are evolved to determine which of the outputs are counted. The circuit that has the largest number of high outputs for an input vector is used to predict the motion for that input vector. All 32 outputs could be counted, even if only 16 of the outputs were used for fitness calculation during the stage 1 evolution.

The best strategy gave an average rate of 73.1% for the training data and 55.1% for the test data. The best run gave 76.33% and 67%, respectively.

3.2 Results by Kajitani

Two articles by Kajitani will be presented here. The first one is “An evolvable hardware chip and its application as a multifunction prosthetic hand controller”[4]. The absolute value of the EMG signal is integrated over a period of 1 second and coded as a 4-bit number. The number of EMG sensors used is not explicitly defined, but since the training pattern length is given as 16 bits it is assumed that 4 sensors are used. The article does not specify whether the data set was generated from a person who had lost a hand or not.

Ten input vectors are made for six different motions, for a total of 60 input vectors. A PLA is used to evolve a solution, then ten additional input vectors are made for the three motions with the lowest fitness. A

new evolution is then performed using the 90 input patterns. The final result was a 81% classification rate averaged over three persons.

The next article is “An Evolvable Hardware Chip for Prosthetic Hand Controller”[5]. A person with no limb loss dons a data glove and two angle sensors, which are used to classify hand actions. Then a frequency spectra of the EMG signal is generated and six frequency bands selected experimentally. Each frequency band is coded using four bits for a total of 24 bits. 10.000 input vectors are generated for each of the 6 motions to be discriminated. 200 of these input vectors are selected for each motion (a total of 1.200) to be used in the evolution, and 800 no-action input vectors are added to the data set. Solutions are implemented using both a neural network and an EHW chip.

The remaining 9.800 input vectors for each motion were used to calculate the fitness of the solutions. 80% was achieved for the neural network, and 85% for the EHW chip. This is not a great improvement over the previous article considering the advanced methods required.

In two additional articles by Kajitani, also using advanced methods, similar results are produced. In one, the average classification rate for 13 persons is 71.2% using an EHW chip, 82.1% using neural networks [26]. In the other, the average classification rate for 5 persons is 85.8% using an EHW chip[12]. The results for the last three articles are not easily comparable to the results in this paper due to the different methods used.

4 Initial results

When starting evolution on the data set, I decided to allow a more advanced circuit design than the two-layer AND-OR method used by Tørresen [23]. This was done in the hope that a more advanced circuit would be able to more accurately pick out the features in the input vector and achieve a higher classification rate. If this proves successful, restraints can then be gradually imposed on the circuit to get an acceptable balance between complexity and performance. The following method is used:

There are n logic blocks $lb_1 \rightarrow lb_n$, each with 2 inputs. For logic block lb_i there are $i + 33$ possible inputs, the constants '0' and '1', the 16 bits in the input pattern, the complement of the 16 bits in the input pattern, and the outputs of any previous logic block ($lb_1 \rightarrow lb_{i-1}$ when $i > 1$). Each logic block can function as an AND, OR, or XOR gate. Two-point crossover is used (see Chapter 2.3). Crossover points are *between* all logic blocks and *after* the last logic block, giving n possible crossover points. When mutation is performed on a circuit, a random logic block is selected, and there is a 25% chance each of input 1, input 2, logic block function, or all three being randomized.

6 different sub-circuits are evolved separately, one for each motion. The output of each sub-circuit is the output of the last logic block lb_n. There are 300 training input vectors iv_j, where $j = \{1, ..., 300\}$. There are also six possible hand/wrist motions, $motion = \{1, ..., 6\}$. For each input vector iv_j there is a corresponding output value $ov_j \in motion$. The six sub-circuits C_m where $m \in motion$ are trained to give a high output for each input vector iv_j that maps to m and a low output otherwise.

$$fitness(C_m) = \sum_{j=1}^{300} x \text{ where } x = \begin{cases} s \text{ if } (C_m(iv_j) = 1 \text{ and } ov_j = m) \\ 1 \text{ if } (C_m(iv_j) = 0 \text{ and } ov_j \neq m) \\ 0 \text{ otherwise} \end{cases}$$

s is a scaling factor used to emphasize input vectors for the motion the sub-circuit is trying to detect. It's effects are shown in Chapter 4.1.

When evaluating the circuit as a whole, each input vector iv and its corresponding output value ov is processed as follows: iv is input into C_1. If C_1gives a high output and $ov = 1$, increase fitness by 1. If C_1gives a high output and $ov \neq 1$, fitness is not increased. In both cases, the process start over with the next input vector. Only if C_1 gives a low output is C_2 tested in the same manner, and so on until C_6 is tested. If all sub-circuits give a low output, no prediction for iv is made, and fitness is not increased.

Roulette selection is used. When selecting individuals for reproduction, the fitness values are scaled using linear scaling (see p. 77 in Goldberg's book[10]) with $C_{mult} = 1.5$ and the resulting negative scaled val-

ues clamped to zero. The remaining parameters vary from run to run, and are:

Generations How many generations, or iterations, the Genetic Algorithm is run.

Runs All numbers and graphs are the results of running each evolution several times and taking the average of all runs. This is the number of individuals that are averaged for each run.

Scaling factor The value of the scaling factor s.

Population_size How many individuals are in the population.

Logic blocks The maximum number of logic blocks the circuit is allowed to use.

Elites How many elites are used. A value of '0' means no elitism is used.

Mutate The chance each logic block in an individual has of mutating after a crossover.

The number of logic blocs is important to the performance of the circuit. If the number of logic blocks is high enough, it becomes possible to evolve a circuit that treats each input vector in the training set as a special case. This will give perfect fitness on the training data, but all generalization will have been lost. This is called specialization, and the result is a training fitness that is much higher than the testing fitness. If the number of logic blocks is too low, it is only possible to evolve simple circuits that will probably give equally good training and testing fitness, but the fitness for both will be too low to be usable. A balance between training fitness and generalization has to be found.

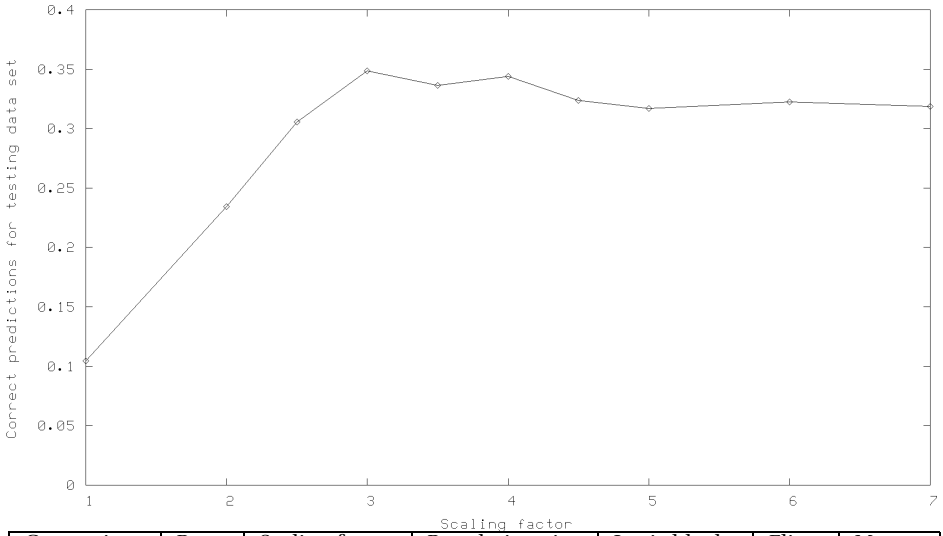
The total number of AND/OR gates in Tørresen's implementation was 32 AND + 32 OR gates, for a total of 64. Depending on the setup, there are two to four inputs per gate. Only two inputs per gate are used in this thesis. The interconnections available are much more advanced however, so 50 logic blocks seems like a reasonable starting choice.

The time needed to perform an evolution should be roughly equal to "Generations" * "Population size" * "Logic blocks", so care should be taken not to use too high values here. Large values of "Logic blocks" and to some degree "Population size" also requires more gates to implement in hardware. Changing the "Mutate", "Scaling factor", and "Elites" values should have little impact on execution time. The initial values were chosen after a few test runs, and are probably not optimal.

I decided to find a good scaling factor first, then vary the number of logic blocks and the population size. The number of elites and the mutation rate to use is then tested, and a conclusion drawn at the end of the chapter.

4.1 Batch 1, scaling factor

In order to determine a good scaling factor, values of s ranging from 1 (no scaling) to 7 are tested. The first run took the average from 10 individuals. Instead of a smooth curve, the values rose and fell, with no clearly defined maximum. To ensure this was not the result of random variations, I decided to perform another run where 100 individuals were averaged instead. The results were the same, there seems to be no smooth curve that can be used to easily pick out the single best value.



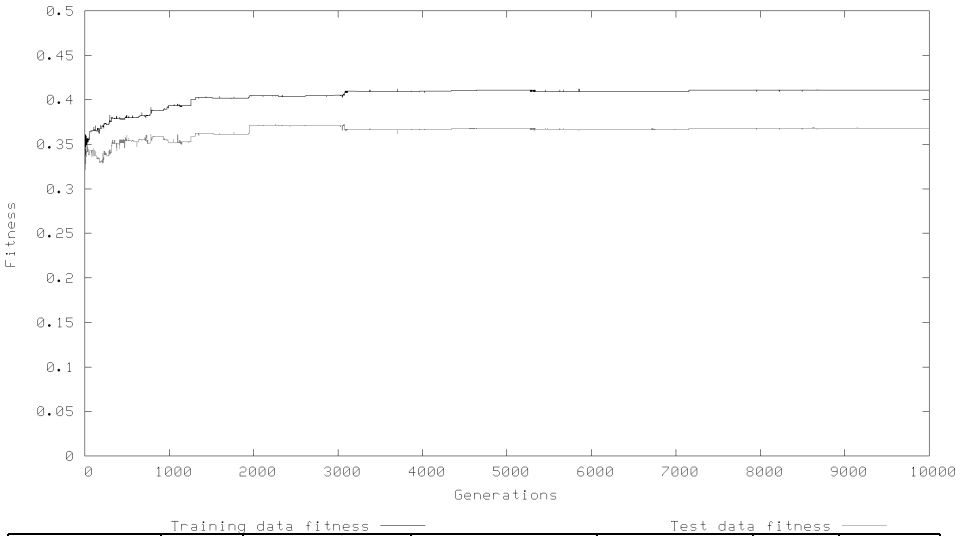
Generations	Runs	Scaling factor	Population size	Logic blocks	Elites	Mutate
1.000	100	Varies	30	50	0	0.01

Figure 10: Determining a good scaling factor

$s = 3$ and $s = 4$ gave the best results, so some additional runs were done with $s = 2.5$, $s = 3.5$ and $s = 4.5$. The results are included in the graph above.

The poor results for values of s less than 3 are caused by the fact that the sub-circuits evolve to give a low output in most or all cases. This gives them a $\frac{5}{6}$ success rate very easily, but evolving the circuit further seems difficult. A circuit with nothing but low outputs represents a local optima that is very easy to find. With few of the sub-circuits giving high outputs for the input vectors they are supposed to identify, the method that combines the six sub-circuits cannot predict the correct motion.

$s = 3$ gave the best result, but looking at fitness for each generation, the fitness does not seem to increase much with the number of generations. Another run with 10.000 generations was performed to illustrate this:



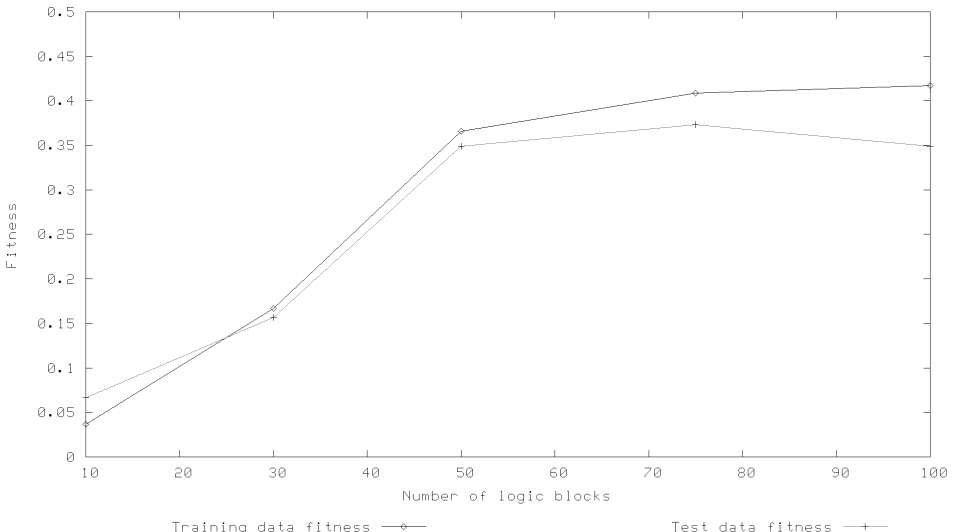
Generations	Runs	Scaling factor	Population size	Logic blocks	Elites	Mutate
10.000	10	3	30	50	0	0.01

Figure 11: 10.000 generations with scaling factor set to 3

The rate of increase is very slow, so there seems little point in increasing the number of generations beyond 10.000. A better set of parameters should be found first.

4.2 Batch 2, logic blocks

The scaling factor of 3 from the last run is used, and the number of logic blocks varied.



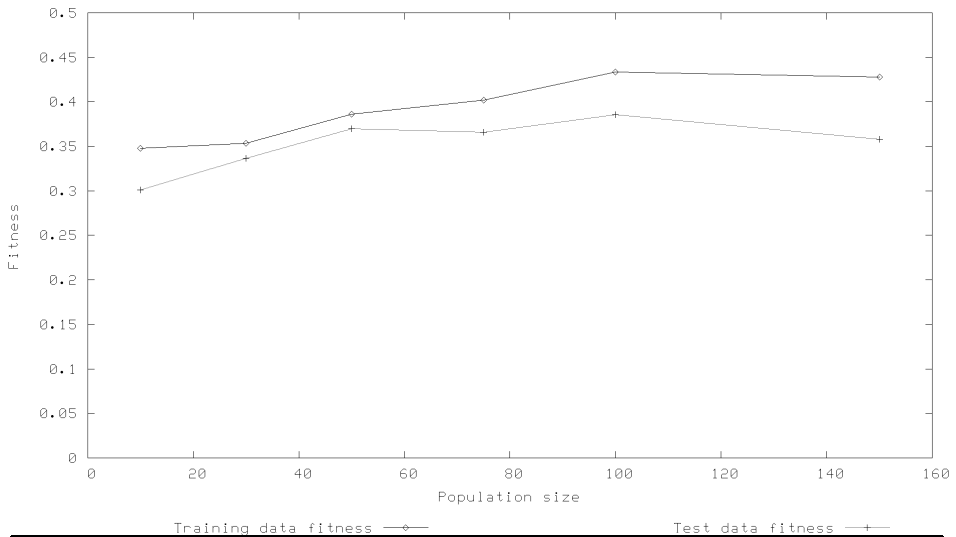
Generations	Runs	Scaling factor	Population size	Logic blocks	Elites	Mutate
1.000	10	3	30	Varies	0	0.01

Figure 12: Determining number of logic blocks

10 and 30 logic blocks seems to be too few to allow a good circuit to evolve. The training fitness increases with the number of logic blocks, but with 100 logic blocks the test fitness decreases. This might be a sign of specialization, but it's hard to say without further data. The difference in the classification rate for 50, 75 and 100 logic blocks is not big, so the number of logic blocks is kept at 50. This should ensure little specialization occurs. The number of logic blocks might need to be increased later, when larger runs are made.

4.3 Batch 3, population size

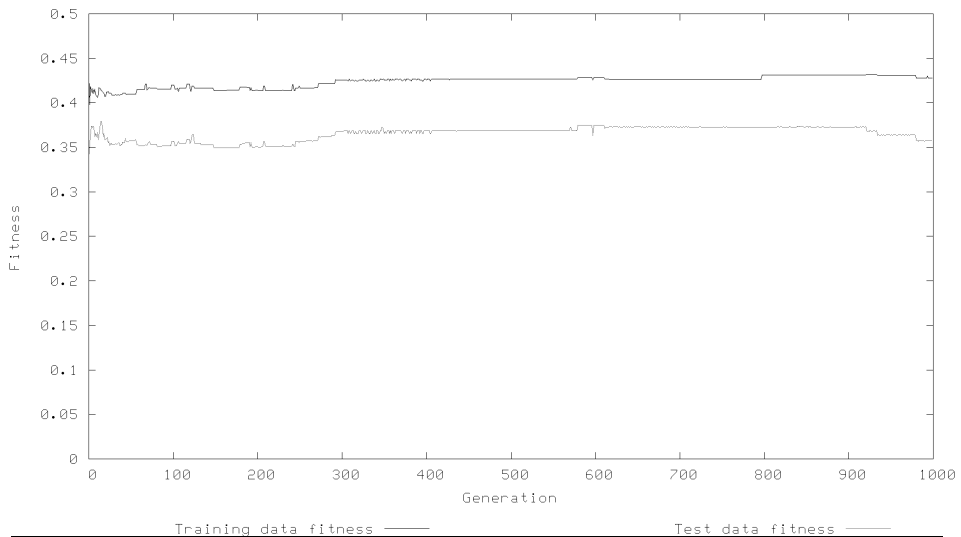
In this batch, the population size is varied.



Generations	Runs	Scaling factor	Population size	Logic blocks	Elites	Mutate
1.000	10	3	Varies	50	0	0.01

Figure 13: Determining population size

A larger population size means that more individuals are evaluated in the course of the entire run, and there is therefore a better chance of finding a good individual. It is therefore a bit surprising that the run with a population size of 150 performed worse than the run with a population size of 100. The fitness chart for the run with a population size of 150 can be seen in Figure 14:



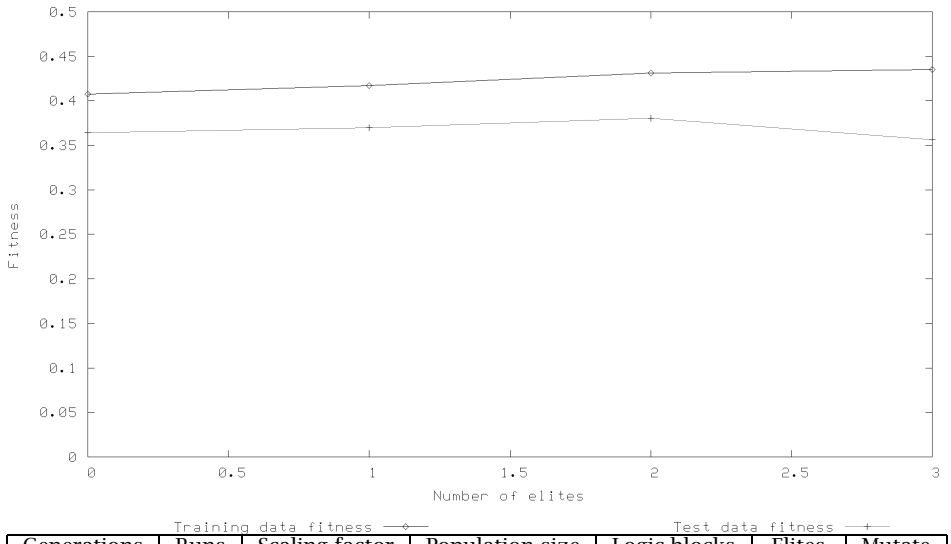
Generations	Runs	Scaling factor	Population size	Logic blocks	Elites	Mutate
1.000	10	3	150	50	0	0.01

Figure 14: The run with a population size of 150

The fact that the fitness values drop from time to time shows that good solutions are being lost. Since there is no chance of cloning, crossover is always performed on every circuit, and a good solution might not survive the crossover. A graph showing the diversity of the population and how destructive the crossover operation is might have been useful here, and is a consideration for future work. Using elites is an easy way to prevent good solutions from being lost, but might give worse long-term performance since the importance of building blocks is decreased. This will be tested in the next batch.

A higher population size gives increased fitness, but also longer execution time. so for a higher fitness the number of generation could be increased instead. I decided to keep the population size at 30 to allow easier comparison of results.

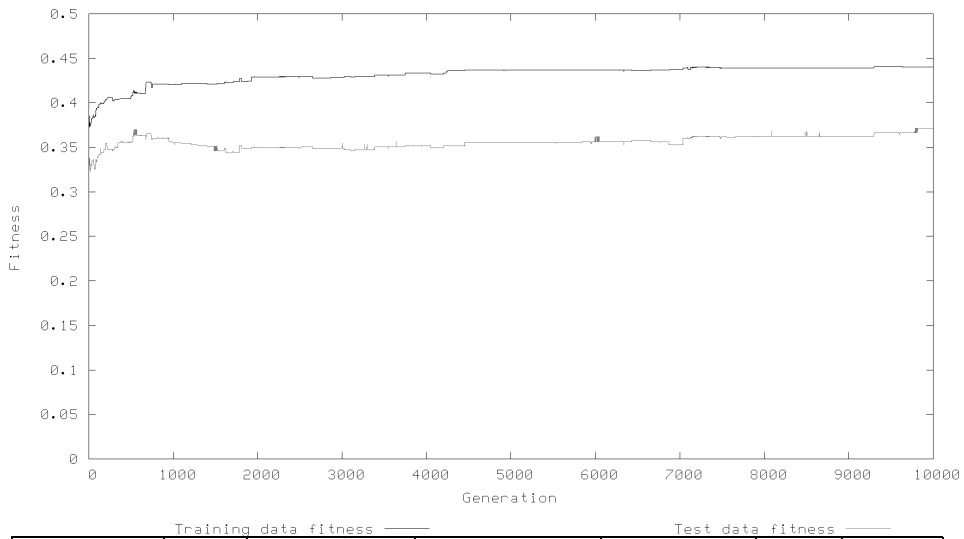
4.4 Batch 4, elites



Generations	Runs	Scaling factor	Population size	Logic blocks	Elites	Mutate
1.000	10	3	30	50	Varies	0.01

Figure 15: Determining number of elites

From the results above, it looks like 2 elites give the best result for training fitness. A run with 10.000 generations and 2 elites was made, so it could be compared to Figure 11.



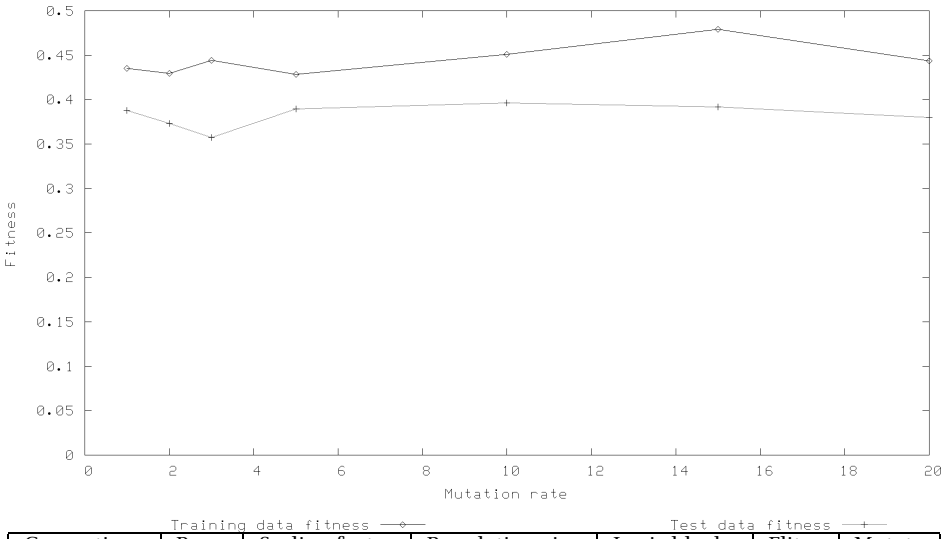
Generations	Runs	Scaling factor	Population size	Logic blocks	Elites	Mutate
10.000	10	3	30	50	2	0.01

Figure 16: 10.000 generations with 2 elites

This shows an slight improvement over the run of 10.000 generations without elites. The end result is better, and there is also some slight increase in fitness towards the end of the run, which means increasing the number of generations should give even better results. I decided to use 2 elites in the following batch.

4.5 Batch 5, mutation rate

Since elites are used in this batch, the mutation rate can be set very high without causing the best individuals to be lost.



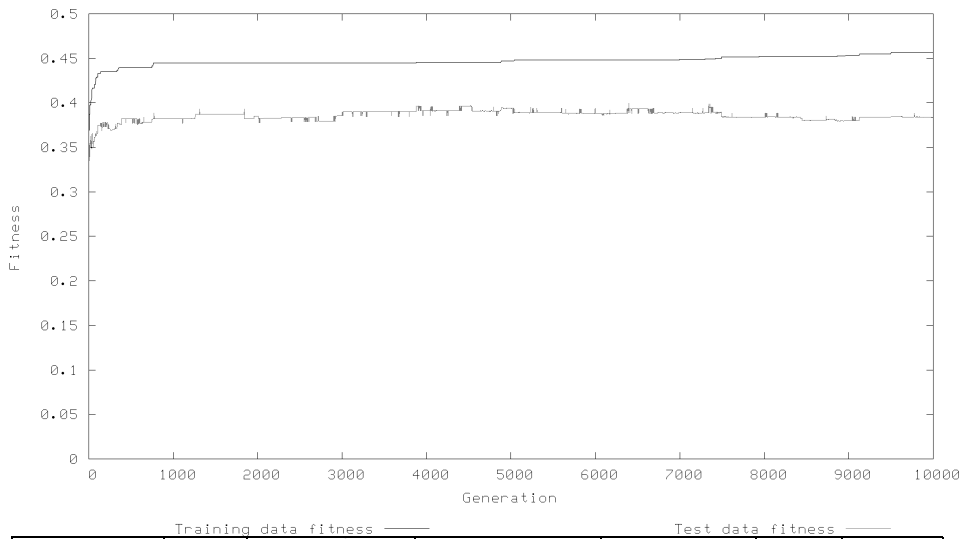
Generations	Runs	Scaling factor	Population size	Logic blocks	Elites	Mutate
1.000	10	3	30	50	2	Varies

Figure 17: Determining mutation rate

A mutation rate of 0.15 gave the best results. With elitism and a high mutation rate, what started out as a genetic algorithm now has a lot in common with a random walk. To determine how much crossover was influencing the search process, I decided to remove the crossover operation completely and do another run with a 0.15 mutation rate.

This gave a training fitness of 0.446 and a test fitness of 0.3787, compared to 0.479 and 0.3916 for the run that used crossover. Crossover is still important to the algorithm despite elitism and a high mutation rate.

Now that the different parameters have all been chose, it's time to see how well the algorithm performs with a higher number of generations.



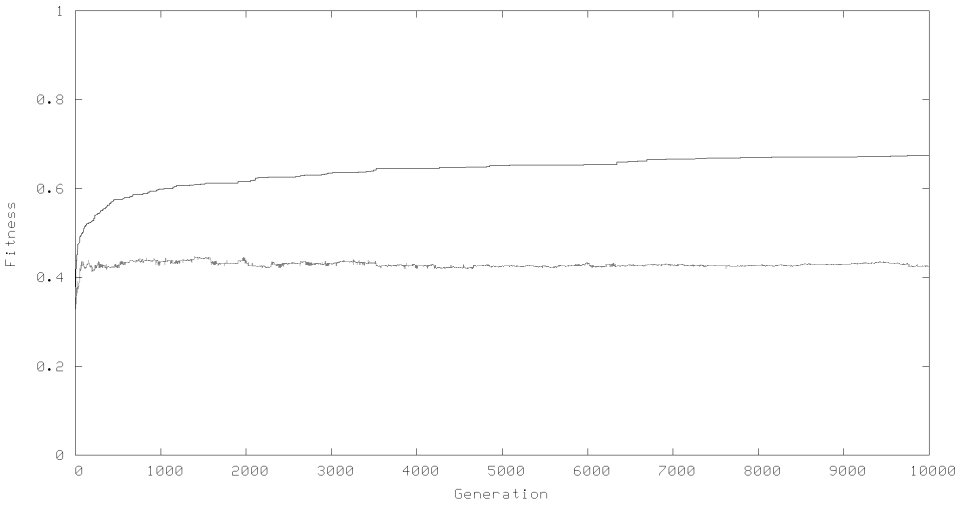
Generations	Runs	Scaling factor	Population size	Logic blocks	Elites	Mutate
10.000	10	3	30	50	2	0.15

Figure 18: 10.000 generation, 0.15 mutation rate

The fitness continues to rise as the number of generations gets closer to 10.000, but the rate of increase is too slow. No matter how many generations the evolution is set to run, it is unlikely the fitness will reach 100%. The limiting factor might be the number of logic blocks. Increasing the number of logic blocks will permit more advanced circuits to be evolved, but might cause specialization.

4.6 Batch 6, maximizing fitness

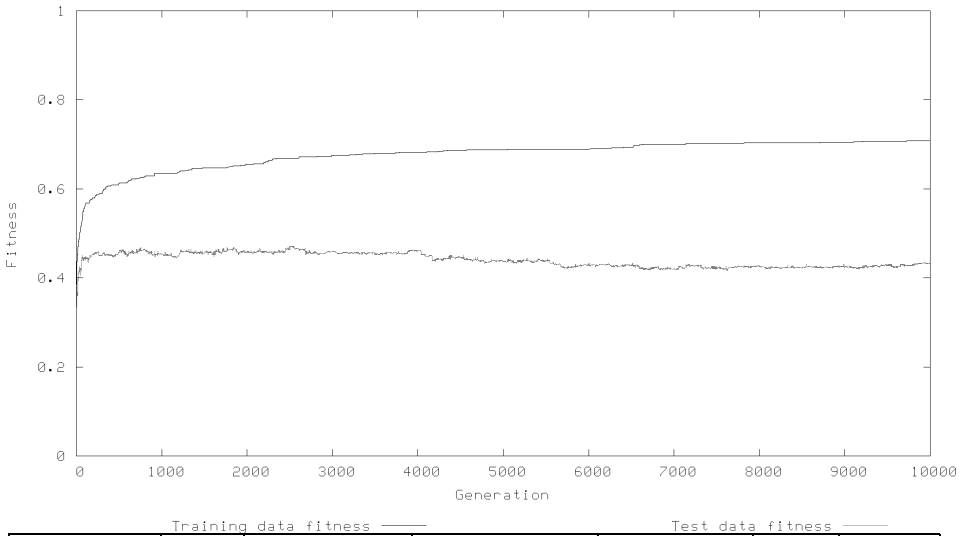
The number of logic blocks is increased to 100 and the performance of a run with 10.000 generations is analyzed.



Generations	Runs	Scaling factor	Population size	Logic blocks	Elites	Mutate
10.000	10	3	30	100	2	0.15

Figure 19: 100 logic blocks

The fitness increased, but now the fitness for the training data is quite a bit higher than the fitness for the testing data. This is a sign of specialization. To explore the connection between the number of logic blocks and specialization, the number of logic blocks is increased to 200.



Generations	Runs	Scaling factor	Population size	Logic blocks	Elites	Mutate
10.000	10	3	30	200	2	0.15

Figure 20: 200 logic blocks

Again fitness has increased, but specialization is readily appearent. With specialization setting in when the training fitness is only at 45%, getting close to 100% training fitness seems impossible. This, together with the fact that a relatively advanced circuit (200 logic blocks * 6, 10.000 generations) is needed to get a 70% training fitness suggests that there might be some limitation in the data set. If this is not the case, the evolution method used might be unsuitable for this data set, and another method needs to be found.

5 Limitations inherent in the data set

5.1 Analyzing the data set

Since the preliminary evolutions indicated an upper limit to the possible classification rate, a large number of parameters were logged during a run and the output analyzed. One thing that was quickly apparent was that there were several input patterns with the exact same input bits. Since there are $2^{16} = 65536$ possible combinations of input bits and only 300 input patterns, this indicates poor spread in the input vectors. This is a problem, since if the input bits for several input patterns which belong to different categories are the same, it becomes impossible to correctly classify all of them. For example, consider the (made up) data in Table 2:

No	Input 1	Input 2	Input 3	Input 4	Category
1	0010	0111	1011	0100	2
2	1001	0111	0110	1100	1
3	1001	0111	0110	1100	1
4	1001	0111	0110	1100	3
5	0011	1111	0100	0101	4

Table 2: A data view of input collision

No. 2-4 have the same input bit pattern, but two of the inputs map to category 1, while the third maps to category 3. The best an algorithm can do is to map that input bit pattern to category 1, meaning that no. 2 and 3 will be correctly mapped, while no. 4 is incorrectly mapped to category 1 instead of category 3.

This problem can be called an *input collision* problem, since the exact same input vector is simultaneously mapped to several different output values. Figure 21 shows how input collision works.

The boxes shows how the input points are quantized. All points within a box are quantized to the value shown in the box, so both category A and category B are quantized to 0,0 and it becomes impossible to distinguish between them. If a more fine-grained quantization had been used, it would have been possible (in theory at least) to distinguish perfectly between them.

Category C and D also have an input collision problem, since points from both categories are quantized to 1,1. In this case there are two differences. Firstly, it is still possible to partly distinguish the categories, since each category contains points outside of 1,1. All points in 2,1 belong to category D, and all points in 0,1, 0,2 and 1,2 belong to category C. Secondly, no matter how fine-grained the quantization becomes, there

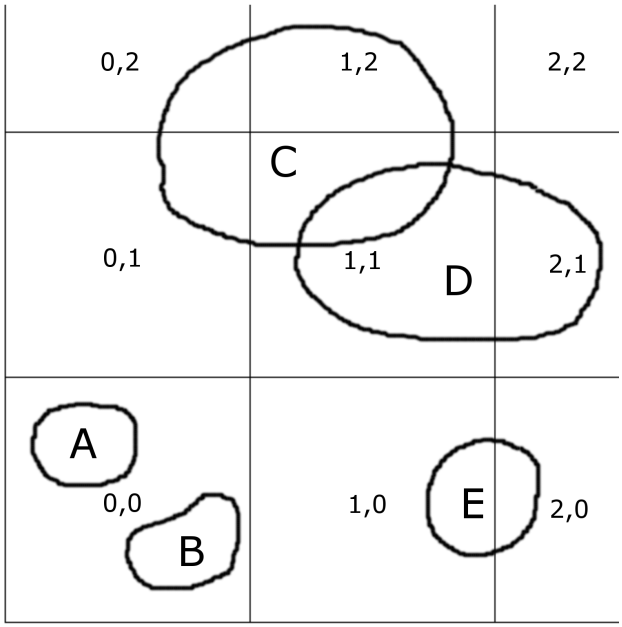


Figure 21: An illustration of input collision

will still be an area where it is impossible to distinguish between the categories. This is the intersection of categories C and D, and can be called *category overlap*.

Category E does not suffer from either input collision or category overlap. All its points are quantized to 1,0 and 2,0, and no other categories have points that are quantized to these values.

Since the only data available is the values produced after quantization, is it impossible to find out how much the input collision problem could be reduced by using another coding scheme. Kajitani has used two different methods to reduce input collision with success, but both these methods use advanced mathematical functions[12, 26]. Since I do not have the equipment to collect another data set myself, and also because the goal is to create a controller that is easily implemented in hardware, I

will try to determine the extent of the problem and find a way to improve the classification rate using other methods.

5.2 The *input collision* problem

To discover the exact number of input vectors that will be incorrectly mapped due to this problem, a special algorithm is needed. The algorithm will, for every unique input bit pattern, assign that pattern to the category which gives the highest possible fitness. This method can be described as follows:

1. Put all input vectors into a set.
2. Remove an input vector from the set.
3. Find all input vectors that have the same bit pattern as the input vector from step 2, and remove them from the set.
4. From the input vectors in steps 2 and 3, find the category which the highest number of input vectors is mapped to.
5. Store in a table the bit pattern together with the category found in step 4.
6. If there are more input vectors left in the set, go to step 2.

Once the table is created, the algorithm will predict which category an input vector belongs to by comparing the input bit pattern with the bit patterns stored in the table, and returning the category stored together with the bit pattern. Since the algorithm in essence makes a look-up table to classify input vectors, it will from now on be referred to as a *look-up algorithm*.

After creating a table from the training data, each value in the training data is passed to the *look-up algorithm* and a record is kept on how many of the values are correctly classified. 270 of 300 inputs were mapped correctly. In other words, the maximum possible success rate is 90%. Creating a table from the test data and passing each value in the test data to the *look-up algorithm* resulted in 261 of 300 correct predictions, or a maximum success rate of 87%.

This algorithm finds the best method of classification for a given data set, but does not generalize at all. To find out the maximum fitness for the test data that could be expected after training with the training data, further tests are needed.

5.3 Finding the maximum test data fitness

Creating a table from the *training* data and using it to predict the categories for each value in the *test* data resulted in only 55 values being predicted correctly. This is not surprising, since many of the input bit patterns in the test data do not exist in the training data, and so the algorithm was unable to map those input bit patterns to a category at all. Therefore, a modification of the *look-up algorithm* is needed.

First, two tables are created, one from the training data (training table), and one from the test data (test table). The modified *look-up algorithm* then predicts which category an input vector belongs to as follows:

1. Compare the bit pattern in the input vector to the bit patterns in the *training* table.
2. If a match is found, return the category stored with the matching bit pattern.
3. Otherwise, return the category stored with the matching bit pattern in the *test* table.

This algorithm emulates an algorithm trained perfectly on the training data, and able to extrapolate perfectly to data not included in the training data. Using the hand prosthetics data sets, the result for the training data set was 270 correct predictions as before, while the result for the testing data set was 209 correct predictions. The success rate is less than 70%.

By further modifying the *look-up algorithm* and weighting the performance on the training data set against the performance on the testing data set, we arrive at the values given in figure 22. These are the maximum attainable results due to the *input collision* problem.

5.4 Some simple algorithms

The results covered so far give an indication of the maximum possible results with full a priori knowledge of the data sets. In a real-world setting, the results will almost always be lower. To gain a rough indication of what results could be expected in the real world, the performance of some simple algorithms is considered. The benefit of using simple algorithms is that they give excellent generalization.

Since these are calculated deterministically, they are of course not evolutionary algorithms. Four different algorithms are considered. They were applied on the training data, and then given first the training data, then the test data as input. An article containing these results was published in Proceedings of 16th European Simulation Multiconference [30].

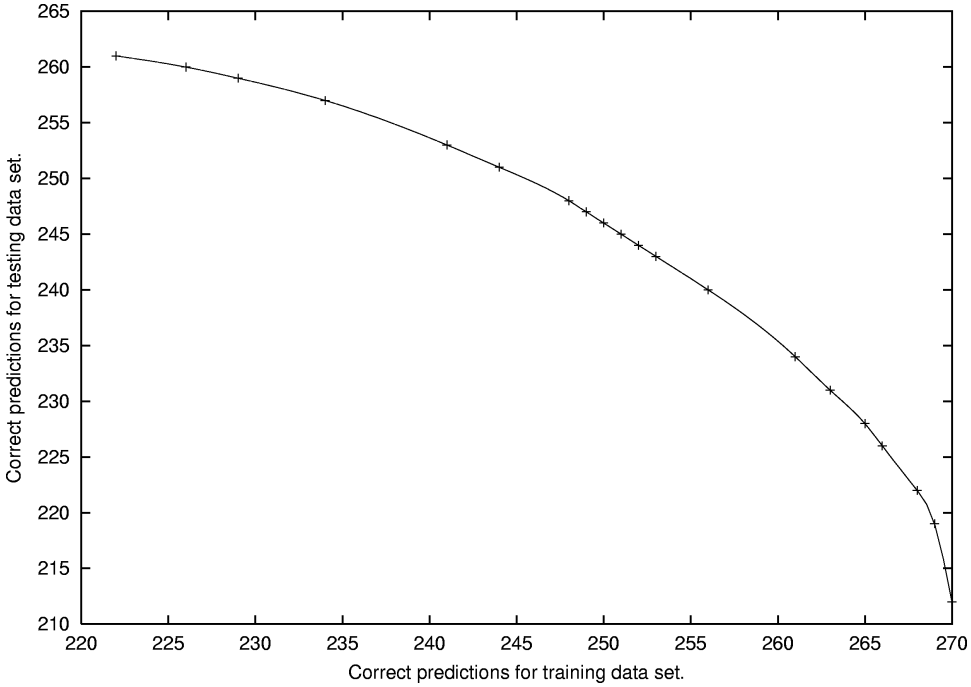


Figure 22: Maximum correct predictions of training vs. testing data.

The pictures used here that illustrate the four methods used were made by Jim Tørresen for a presentation he held, and are used here with his permission.

All of the algorithms presented here use the concept of distance. For a given value, its distance to an input vector is equal to the number of bits that are different between the two. The distance between 100 and 010 is 2, since two bits are different. Each input vector also has a motion associated with it. Some of the algorithms require us to sum the motions for all input vectors within a set distance. In Table 3, an example is given using 8 input vectors. The value we are measuring the distance to is 100.

In the descriptions below, when a referral is made to the motion which occurred the most times, if two or more motions tie for highest occurrence, the first one is used.

Bit pattern	# bits different	M1-M6	Sum M1-M6
100	0	001000	001000
101	1	100000	
110	1	001000	
000	1	000010	102010
111	2	001000	
001	2	010000	
010	2	100000	213010
011	3	000100	213110

Table 3: The concept of bit distance.

Smallest distance: Find the input vectors with the lowest number of bits different from the value we are testing and return the motion with the highest occurance.

Data set	No correct
Training data	270/300
Test data	151/300

Average: Find all input vectors with N or less bits different from the value we are testing and return the motion with the highest occurance.

Data set	N=0	N=1	N=2	N=3	N=4	N=5	N=6	N=7
Training data	270	222	195	185	162	151	128	121
Test data	151	157	162	169	160	157	156	144

First N vals: Start by summing the motions for all input vectors with 0 bits different from the value we are testing. If one of the motions occured N or more times, return that motion. If not, sum the motions for all input vectors with 1 or 0 bits different, and so on.

Data set	N=1	N=2	N=3	N=4	N=5	N=6	N=7	N=8	N=9
Training data	270	259	245	237	223	198	197	187	187
Test data	151	150	154	152	152	166	161	156	160
Data set	N=10	N=11	N=12	N=13	N=14	N=15	N=16	N=17	
Training data	179	172	172	173	173	170	166	164	
Test data	165	169	169	172	171	170	168	167	

First N uniques: This method is a bit more advanced. It starts by looking at the sum of motions for all input vectors with 0 bits different from the value we are testing. If one motion occurs more times than any other motion, one point is given to that motion. This is then repeated with the sum of motions for all input vectors with exactly 1 bit different from the value we are testing, and so on. When a motion gets N points, return that motion.

Data set	N=1	N=2	N=3	N=4	N=5	N=6	N=7
Training data	270	230	217	183	186	186	186
Test data	152	174	173	168	166	166	166

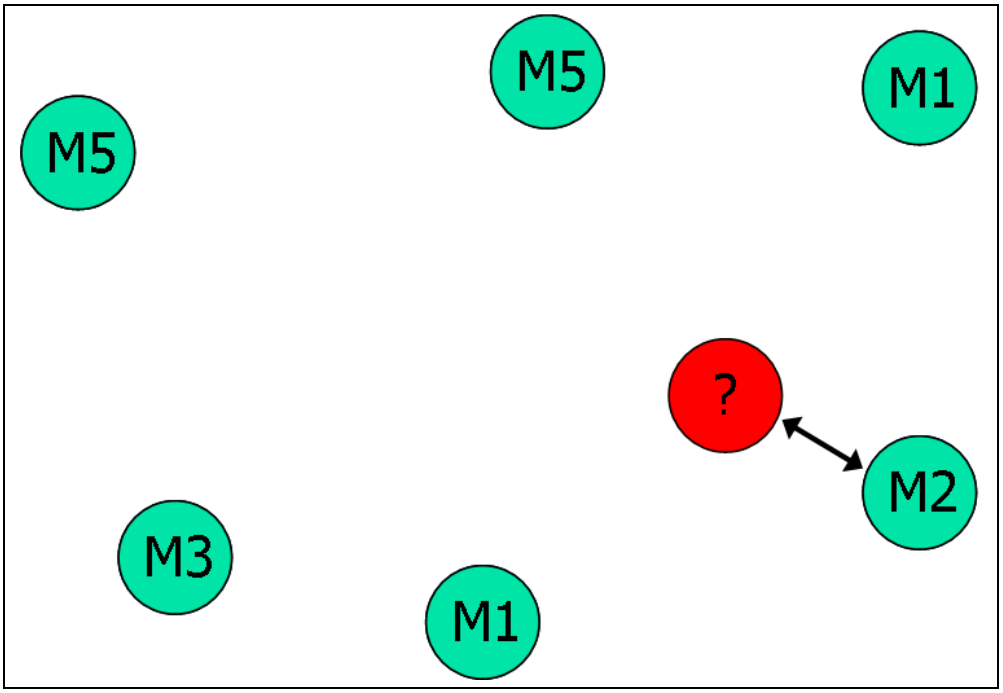


Figure 23: Smallest distance

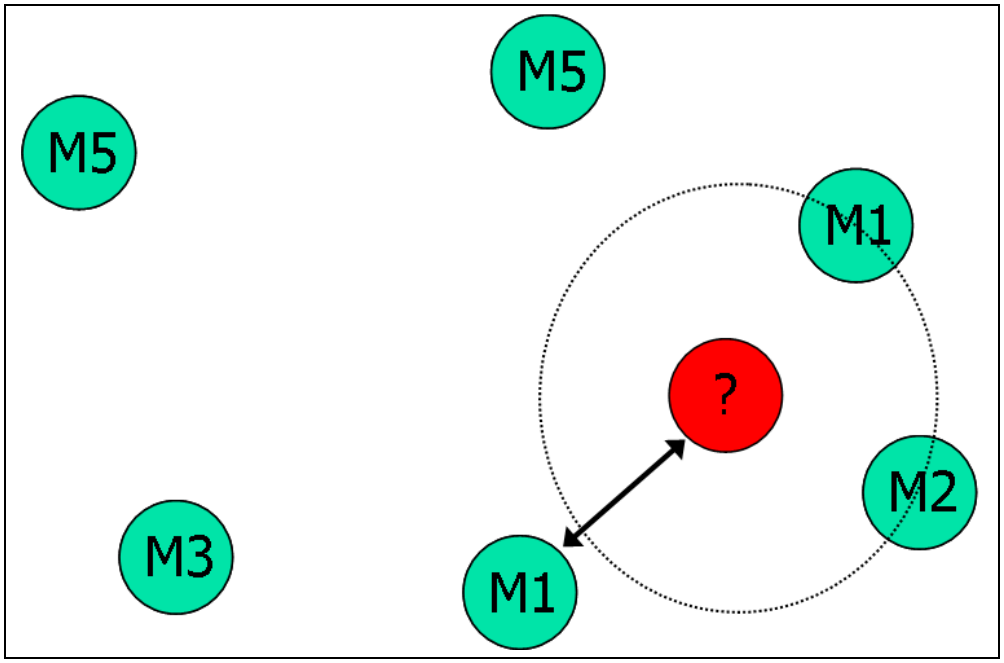


Figure 24: Average

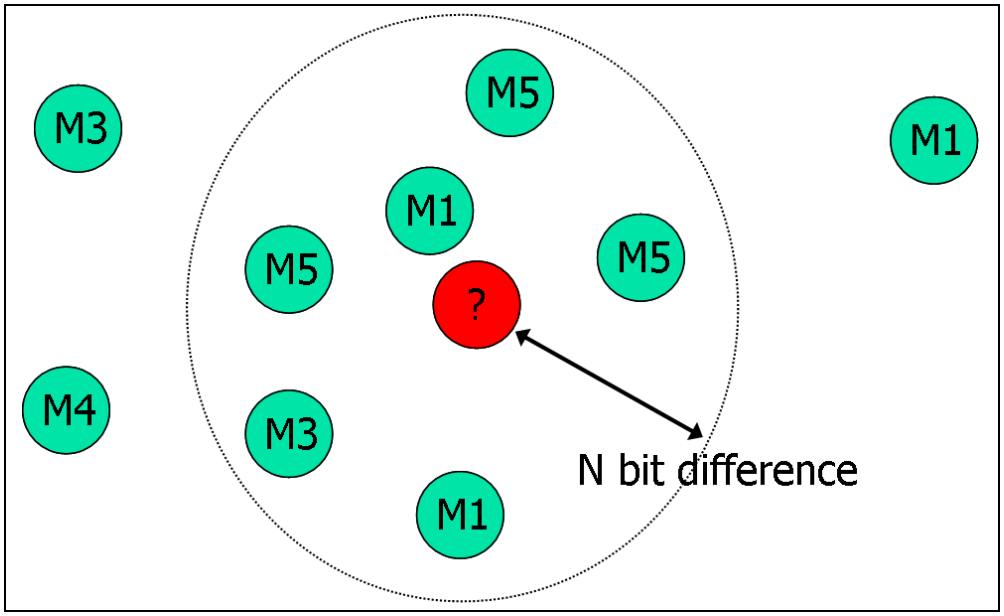


Figure 25: First N vals

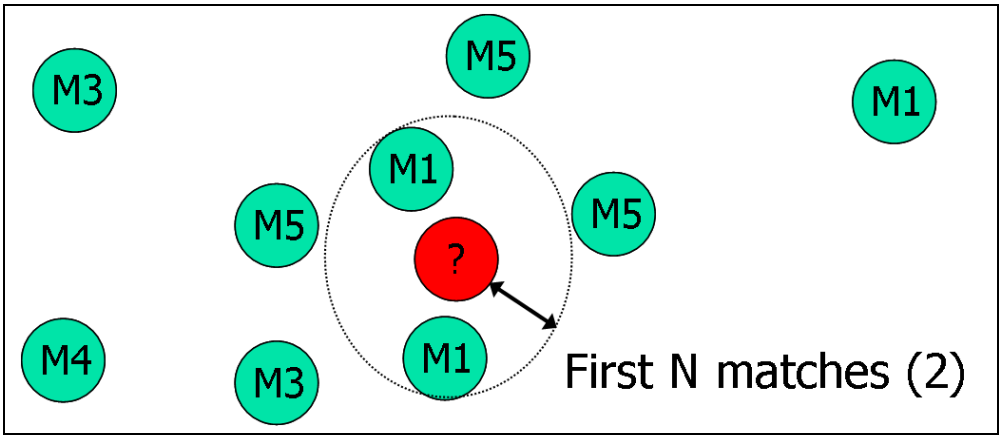


Figure 26: First N uniques

The data is summarized in figure 27. There is quite a large gap between the performance of these algorithms and the maximum attainable performance. A good evolutionary algorithm would most probably perform better than the deterministic algorithms described here, but below the maximum attainable performance.

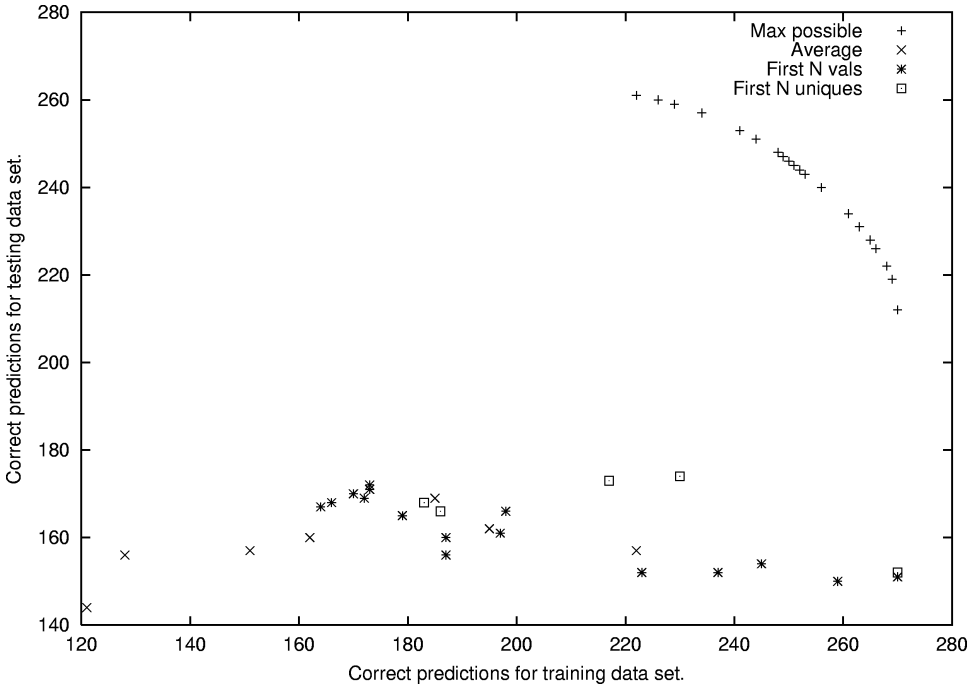


Figure 27: Performance of deterministic algorithms.

6 Improving the data set

Based on the results uncovered in the previous chapter, getting even a 75% correct classification rate in a real-world environment seems optimistic. Surpassing the performance of the circuit evolved in Tørresen's work[23] might be possible, but any improvement will most likely be slight at best. To get higher performance, the limitations in the data set need to be circumvented. This chapter will consider different methods of doing this.

Since the problem lies in the data set, the only way to remove the problem is to modify or replace the data set. There are several different ways this can be done:

1. Increasing the bit length of the input values.
2. Collecting a new data set using a different preprocessor method.
3. Using several sample values to predict the output.

6.1 Increasing the bit length of the sample values

The input collision problem could probably be reduced considerably by doubling the length of the input bit patterns from 16 to 32 bits. The first problem with this approach is that it requires more advanced hardware for implementation. The second problem is that this will drastically increase the search space, meaning that it might take a lot longer to evolve an acceptable circuit.

One way of increasing the bit length is to use 8 bits per channel instead of 4. However, this will not reduce category overlap, only input collision. Another way of increasing the bit length is to use 8 EMG sensors instead of 4. However, additional EMG sensors would increase the cost and complexity of the prosthesis.

In conclusion, increasing the bit length without making other changes to the data does not seem like a good solution.

6.2 Collecting a new data set using a different method

The integrating approach used when collecting the data set was used because it required a lot less processing than for example Fourier transforms. If we wish to avoid complex computations, the options for collecting a data set become very limited.

Instead of integrating, the maximum, minimum, or median value could be used, but this seems unlikely to have a positive effect on the

classification rate. The integrated value could be quantized differently, as was done by Kajitani et al in recent works, but this requires more hardware[12, 26]. 2 channels with 8 bits each could also be used. This might help with the input collision problem, but would not reduce category overlap.

Adjusting the length of each integrating period is another option, but integrating for too long a time period will slow down the response time of the prosthetic arm, and shortening the integrating period is unlikely to yield better data.

In conclusion, modifying the encoding and varying the integrating period might be worth trying, but does not seem likely to yield significantly better results.

6.3 Using several sample values to predict the output

Instead of predicting the output by using only one input vector, several input vectors could be combined to predict the output. The first problem with this approach is that requiring several input vectors slows down the response time of the prosthetic. This could be compensated for by generating several input vectors in the same time span, but since each input vector then represents a shorter time span, the classification rate on each input vector might degrade, negating any benefit from combining the input vectors.

One way of using several input vectors would be to simply combine two 16-bit input vectors into a single 32-bit input vector. The two problems of requiring more advanced hardware and increasing the search space still apply. But it might be a better solution than the idea of adding 4 more channels, since this method does not have the increased complexity resulting from 4 additional EMG sensors. It does have the problems listed above however, so it is a trade-off between different problems.

Another solution is to evaluate each input vector separately, then combine the output from each evaluation into a final result. This could be done without the problems resulting from increasing the input bit length, but would require additional logic to combine the results. In addition, the circuit would either operate slower, since it needs to evaluate several input vectors in serial, or require even more logic to evaluate several input vectors in parallel or with caching.

Evaluating several input vectors separately and combining the outputs into a final result seem like the best method to try. But the improvements need to be considerable for this method to be worthwhile. The integrating period should be shortened to compensate for the additional input vectors needed, and the number of generations should be

limited to compensate for the extra execution time needed if evaluating the input vectors in serial.

6.4 Modifying the existing data set

From the different methods described above, there are only four methods that do not require complex hardware:

1. Substituting maximum/minimum/median for integration when generating samples. (Section 6.2)
2. Using more or fewer channels with more/fewer bits per channel. (Sections 6.1 and 6.2)
3. Changing the integrating time used for each sample. (Section 6.2)
4. Evaluating several input vectors separately and combining the outputs into a final result. (Section 6.3)

From these, the best option seems to be no 4. One of the benefits is that the same data set can be used, making it easy to compare the results to the earlier results given here. The main problem is that to achieve the same response time for the prosthetic hand, the integrating time used for each sample must be reduced. When comparing the results, this must be kept in mind.

As described in Section 3, 10 input vectors are collected immediatly following each other. These 10 input vectors can be called a *series*. This means that there are 10 series for each motion, 5 in the training data set and 5 in the test data set. When combining input vectors, it makes sense to use those that are sampled at almost the same time, since that is what a control system in a prosthesis has to do. If the control system has to react within 500 ms, and needs 5 input vectors to predict a motion, all 5 input vectors have to be sampled within the 500 ms time window.

For example, when combining 3 input vectors, they will be combined as follows:

Complete series: 1-2-3-4-5-6-7-8-9-10

Possible combined input vectors: 1-2-3, 2-3-4, 3-4-5, 4-5-6, 5-6-7, 6-7-8, 7-8-9, 8-9-10

Thus, the number of combined input vectors available per series is 8 when using this method. Mathematically, the number of combined input vectors available can be described as:

$no_{civ} = no_{series} * (no_{series\,length} - no_{combine} + 1)$

where

no_{civ} is the number of combined input vectors that will be available.

no_{series} is the number of series in the data set (in this case 30).

$no_{series\,length}$ is the number of input vectors in each series (in this case 10).

$no_{combine}$ is the number of input vectors to combine (3 in the example above).

So using 1 input vector to calculate the final fitness gives $30 * (10 - 1 + 1)$ or 300, which corresponds to the 300 input vectors available in the training data set. Combining 3 input vectors gives $30 * (10 - 3 + 1)$ or 240 combined input vectors. Combining 7 input vectors gives $30 * (10 - 7 + 1)$ or 120 combined input vectors. As the number of input vectors to be combined increases, the number of values available to calculate the fitness decreases together with the accuracy of the result. 120 combined input vectors should still give sufficiently accurate information, but combining more than 7 samples is probably not a good idea with the data available.

7 Results from using the modified data set

The individual sub-circuits are evolved using the 300 (not combined) input vectors in the training set. This is the same method as used earlier. The difference is in how the sub-circuits are combined. Instead of using a single input vector and predicting the motion from the first sub-circuit that gives a high output signal, several input vectors are used. Each input vector is input into each sub-circuit, and the number of times each sub-circuit gives a high output signal is recorded. Once all input vectors have been processed, the sub-circuit which gave a high output signal for the most input vectors is the winner, and the motion corresponding to that sub-circuit is used.

The results will be shown in the following format:

Parameters					
Generations	Scaling factor	Population size	Logic blocks	Ratio	Mutate

Training data								
	Combine	1	2	3	4	5	6	7
	Run x							
	Average							

Testing data								
	Combine	1	2	3	4	5	6	7
	Run x							
	Average							

Table 4: Example table

The training and testing data are divided into two sections. In each section, the number of input vectors that are combined when evaluating the final fitness is shown in the “Combine” row at the top. The individual results from each run are shown in seperate rows, and the average from the runs is shown at the bottom. In some cases only the average is included, with the number of runs indicated in parenthesis. Some test runs showed that a higher scaling factor gave a better result when combining input vectors. A scaling factor of 5 is used for all the results in this section. All other parameters are the same as the ones used in Section 4.

Instead of presenting all the results from evaluating each parameter as was done earlier, only the best results are shown.

Parameters					
Generations	Scaling factor	Population size	Logic blocks	Elites	Mutate
100	5	30	30	0	0.01

Training data							
Combine	1	2	3	4	5	6	7
Run 1	46%	49.63%	49.58%	52.86%	52.77%	54%	51.67%
Run 2	40%	48.52%	52.08%	53.33%	58.33%	56%	57.5%
Run 3	36%	47.04%	52.92%	57.62%	62.78%	66.67%	65.83%
Run 4	37.67%	44.44%	45.42%	46.19%	48.89%	48%	49.17%
Run 5	35.33%	42.22%	45.42%	47.14%	46.67%	46.67%	45.83%
Average(5)	39%	46.37%	49.08%	51.43%	53.89%	54.27%	54%

Testing data							
Combine	1	2	3	4	5	6	7
Run 1	31.67%	45.19%	45.83%	49.52%	51.67%	52%	51.67%
Run 2	37%	37.04%	33.75%	31.43%	30.56%	29.33%	29.17%
Run 3	39%	44.81%	50.42%	56.19%	60.56%	62.67%	61.67%
Run 4	42%	42.96%	41.25%	41.43%	42.78%	43.33%	44.17%
Run 5	36%	40.74%	44.17%	46.67%	47.78%	47.33%	46.67%
Average(5)	37.13%	42.15%	43.08%	45.05%	46.67%	46.93%	46.67%

Table 5: Input value combining

Looking at these results, there is a marked increase in the classification accuracy when combining input vectors. The most marked increase occurs when going from no input vector combining to combining 2 input vectors. This increase occurs in both the training and testing data. Combining more than 2 input vectors causes a fitness increase in some cases and a fitness decrease in other cases. The optimal value in this case seems to be 5 or 6, with 7 showing a drop in fitness. It must be remembered that the accuracy of the data decreases as the number of combined input vectors increase, since there are fewer test cases to calculate fitness from. More data is needed before drawing any conclusions.

Parameters					
Generations	Scaling factor	Population size	Logic blocks	Elites	Mutate
10.000	5	30	30	0	0.01

Training data							
Combine	1	2	3	4	5	6	7
Average(5)	41.87%	49.48%	55.08%	60.48%	63.33%	64%	66.33%

Testing data							
Combine	1	2	3	4	5	6	7
Average(5)	36.73%	42.22%	46.25%	49.43%	52.44%	52%	52.33%

Table 6: Input value combining, 10.000 generations

When increasing the number of generations to 10.000, there is only a slight increase in the Combine 1 column, but the subsequent columns (2-7) show a relatively much larger increase. A small increase in the

fitness for a single input vector causes a much larger increase in the fitness for combined input vectors. This means that if other ways of increasing fitness can be found, such as removing the input collision problem, it should be easy to get a 100% classification rate for combined input values.

Parameters					
Generations	Scaling factor	Population size	Logic blocks	Elites	Mutate
10.000	5	30	100	0	0.01

Training data							
Combine	1	2	3	4	5	6	7
Average(5)	46.07%	54.67%	62.67%	68.67%	73%	76.8%	77.67%

Testing data							
Combine	1	2	3	4	5	6	7
Average(5)	44.4%	49.78%	55.92%	62.86%	67.78%	71.07%	71.17%

Table 7: Input value combining with 100 logic blocks

Increasing the number of logic blocks to 100 had a large effect on the fitness numbers. The fitness for the testing data especially improved greatly. Comparing 7 combined input vectors with no combined input vectors for the first three runs shows the usability of this method. In the first run, the classification rate for the training data increased from 39% to 54% (15%), in the second run it increased from 41.87% to 66.33% (24.46%), and in the third run it increased from 46.07% to 77.67% (31.6%). A 7.07% increase in the classification rate for no combined input vectors led to a 23.67% increase in the classification rate for 7 combined input vectors.

The number for the testing data are 37.13% to 46.67% (9.54%), 36.73% to 52.33% (15.6%), and 44.4% to 71.17% (26.77%).

Some experimentations with 200 logic blocks showed that 1 elite and a 0.02 mutation rate gave better results than the values used in the last run, so these values are used here.

Parameters					
Generations	Scaling factor	Population size	Logic blocks	Elites	Mutate
10.000	5	30	200	1	0.02

Training data							
Combine	1	2	3	4	5	6	7
Average(5)	69.53%	84.15%	92.08%	97.33%	97.89%	98.93%	99.5%

Testing data							
Combine	1	2	3	4	5	6	7
Average(5)	45.8%	50.81%	59.25%	63.14%	66.11%	67.33%	67.83%

Table 8: Input value combining with 200 logic blocks

With the large increase in fitness for the training data, the fitness for combining input vectors rises to almost 100%. Some sort of threshold seems to have been passed, as the fitness now rises continuously all the way to 7 combined input vectors. This is interesting, since it means that creating a system with more than 7 combined input vectors might cause the fitness values to continue rising. Since specialization is setting in, the fitness for the testing data did not improve.

The testing fitness does not increase much when using a large number of generations, so in the following run the number of generations is set to 100. The other values were determined through some experimentation, and the result is an algorithm that gives a good testing fitness with little execution time.

Parameters					
Generations	Scaling factor	Population size	Logic blocks	Elites	Mutate
100	5	200	200	1	0.1

Training data							
Combine	1	2	3	4	5	6	7
Best run	58.33%	71.85%	85.83%	91.90%	92.78%	94.67%	94.17%
Average(500)	54.14%	65.52%	75.04%	82.1%	86.87%	90.23%	92.25%

Testing data							
Combine	1	2	3	4	5	6	7
Best run	48.67%	62.92%	69.58%	79.52%	88.33%	92%	96.67%
Average(500)	44.41%	52.92%	58.78%	64.64%	68.60%	70.42%	71.18%

Table 9: Input value combining, 100 generations

The training fitness is not very good without using input value combining, but when combining 7 input values the results are good indeed. This shows that a good solution can be evolved very quickly. The large number of runs (500) means the values given here should not be far from the true average performance of this method.

The single run included is the run that gave the best testing fitness. This shows that it is possible to evolve almost perfect circuits. But in practice it will be almost impossible to get a training fitness that high. The much lower averages are still a significant improvement over the results gained earlier, and show that combining several input vectors is definitely a viable option.

Since the fitness rises with the number of input vectors combined, it should be possible to combine even more input vectors to get even better results. The number of combined input vectors can be selected to give the best trade-off between fitness value and response time of the prosthetic.

Extra logic is needed to take the results from evaluating several input vectors and combining those results into a final value, but the operation

is simple and seems to be well worth the effort. Compared with Tørresen's work[23], which reported average values for the best method of 73.1% for the training data and 55.1% for the test data, combining 7 input vectors achieve 92.25% and 71.18% respectively.

8 Conclusion

When trying to evolve a good solution to the problem of classifying integrated EMG signals, no acceptable solutions were found. Upon analyzing the data set, it was discovered that the integrating method used for collecting the input vectors gave a poor spread of encoded values, and an *input collision* problem occurred. The problem was analyzed and the maximum fitness values were found. These were considered too low, and a way to circumvent this problem by modifying the data set was considered. Combining input vectors was considered to be the best alternative, and new evolutions were run using this method.

By combining input vectors, it was possible to raise the test fitness average from 44.41% to 71.18%. A earlier paper using the same data set was able to achieve a test fitness average of 55.1%. Combining input vectors has the big disadvantage of slowing down the response time, so this method is dependent upon being able to decrease the integrating time. Since the combined input vectors were taken from a set that contained only 10 values per time series, it is hoped that increasing the number of values sampled in each time series will cause the fitness to rise more as the number of combined input vectors are increased, but there is no guarantee that this will actually be the case.

The combining method is simple to implement in hardware, and should be well suited to an on-line EHW system. By caching the results from evaluating input vectors, a hardware system using combined input vectors should be able to run at the same speed as an equivalent system with no input vector combining. However, more work needs to be undertaken to determine the minimum length of time an EMG signal can be integrated and still give usable results. A decrease in fitness due to a shorter integrating period is acceptable only if the higher number of combined input vectors compensates for this.

9 Future work

The integrating period for the input vectors in the data set used here is very long. One second per input vector means a very long time will pass from the user contracts a muscle til the prosthetic hand controller is able to analyze the signal. When combining input vectors, the problem becomes even worse. 1.6 seconds is needed when combining 7 input vectors. A shorter integrating period would be beneficial, since the response time could be shortened and more input vectors could be combined.

Reducing the number of EMG sensors to 2 would also be beneficial, since the controller could be made cheaper and less complex. Since the input collision problem is pretty big with 4 EMG sensors, using 2 sensors with 8 bits per sensor instead of 4 sensors with 4 bits per sensor should not make much of a difference. It might even improve the situation.

The method of combining input vectors should work well with any set of input vectors. Therefore finding a way to prevent input collision should enable even higher classification rates. This might however be difficult to do in a way that is easily implemented in hardware. Since the best average training fitness achieved here was 45.8% and another work using the same data set achieved 55.1%, there is also room for improvement in the methods used to evolve circuits.

Since a single impulse to a muscle is only a few ms long, it seems resonable to assume that the minimum amount of time the EMG signal must be integrated to get useful data is between 10 and 100 ms. The total amount of time needed to predict a single motion is dependent on three factors: The amount of time each input vector is integrated, the time between integration of one input vector begins and integration of the next input vector begins, and the number of input vectors that are to be combined. There will likely be a trade-off between the amount of time spent and the classification rate at each step. The three factors need to be balanced against each other to give the highest possible classification rate within a given time period.

All the suggestions listed here require the collection of more data. The main consideration for future work is therefore getting equipment to measure EMG signals, preferrable with a high enough sampling rate that a single data set can be made and analyzed digitally. If the sample rate is high enough, integration can be performed digitally, and the same data set can be used when comparing different values for the three factors mentioned above.

References

[1] B. Hudgins, P. Parker and R. N. Scott. “A new strategy for multifunction myoelectric control”, *IEEE transactions on biomedical engineering* 40(1):82-94. 1993.

[2] S. Schultz, C. Pylatiuk and G. Bretthauer. “A new ultralight anthropomorphic hand”, *Proceedings of the 2001 IEEE International Conference on Robotics & Automation*, Seoul, Corea, volume 3, pp. 2437 -2441, May 2001.

[3] M. C. Carrozza, P. Dario, R. Lazzarini et al. “An actuator system for a novel biomechatronic prosthetic hand”, *Proceedings of Actuator 2000* Bremen, Germany, pp. 276-280. 2000.

[4] I. Kajitani, T. Hoshino, N. Kajihara, M. Iwata and T. Higuchi. “An evolvable hardware chip and its application as a multifunction prosthetic hand controller”, *Proc. of 16th National Conference on Artificial Intelligence* (AAAI-99), 1999.

[5] I. Kajitani, M. Murakawa, D. Nishikawa, H. Yokoi, N. Kajihara, M. Iwata, D. Keymeulen, H. Sakanashi and T. Higuchi. "An Evolvable Hardware Chip for Prosthetic Hand Controller". *Proc. of the Seventh International Conference on Microelectronics for Neural, Fuzzy, and Bio-Inspired Systems* (MicroNeuro99), pp. 179-186, 1999.

[6] A. Thompson. “An evolved circuit, intrinsic in silicon, entwined with physics”, *Proceedings of 1st Int. Conf. on Evolvable Systems* (ICES’96), LNCS. Springer-Verlag, 1996.

[7] P. Dario, M. C. Carrozza, S. Micera, B. Massa and M. Zecca. “Design and Experiments on a Novel Biomechatronic Hand”, *ISER 2000*, pp. 159-168. 2000.

[8] M. P. LaPlante and D. Carlson. “Disability in the United States; Prevalence and Causes”, Report No. 7, Disability Statistics Center, University of California, San Fransisco, Jan. 1996.

[9] S. Fujii, D. Nishikawa and H. Yokoi. “Development of prosthetic hand using adaptable control method for human characteristics”, *IAS-5*, pp. 360-367. 1998.

[10] D. E. Goldberg. “Genetic Algorithms in Search, Optimization, and Machine Learning”, ISBN 0-201-15767-5. 1989.

[11] W. Banzhaf, P. Nordin, R. E. Keller and F. D. Francone. “Genetic Programming, An Introduction”, ISBN 1-55860-510-X. 1998.

[12] I. Kajitani, N. Otsu and T. Higuchi. “Improvements in Myoelectric Pattern Classification Rate with μ-law Quantization”, *Proceedings of the XVII IMEKO World Congress*, Vol. 2, pp. 2032-2035, 2003.

[13] A. H. Arieta, W. Yu, M. Maruishi, H. Yokoi and Y. Kakazu. “Integration of a Multi-D.O.F. Individually Adaptable EMG Prosthetic System with Tactile Feedback”.

[14] K. Ito, T. Tsuji, A. Kato and M. Ito, M. “Limb-function discrimation using EMG signals by neural network and application to prosthetic forearm control”, *Proceedings of the IJCNN91*, pp. 1214-1219. 1991.

[15] D. H. Silcox, M. D. Rooks et al. “Myoelectric Prostheses”, *The Journal of Joint and Bone Surgery*, Vol 75-A, No 12, pp. 1781-1791, 1993

[16] J. Tørresen. “Possibilities and Limitations of Applying Evolvable Hardware to Real-World Applications”, *10th international conference on Field Programmable Logic and Application* (FPC-2000). Villach, Austin, Aug 2000.

[17] J. F. Miller, D. Job and V. K. Vassilev. “Principles in the Evolutionary Design of Digital Circuits - Part 1. Genetic programming and evolvable machines 1(1/2):7-35”, 2000.

[18] X. Yao. “Promises and Challenges of Evolvable Hardware”, *IEEE transactions on systems, man and cybernetics - Part C: Applications and reviews* Vol 29 No 1, pp, 87-97, 1999.

[19] M. F. Kelly, P. A. Parker and R. N. Scott. “The application of neural networks to myoelectric signal analysis: A preliminary study”, *IEEE transactions on biomedical engineering* 37(3):221-230. 1990.

[20] D. Howe, ed. “The Free On-line Dictionary of Computing”, http://www.foldoc.org/.

[21] J. Heitkoetter and D. Beasley, eds. (2001) “The Hitch-Hiker’s Guide to Evolutionary Computation: A list of Frequently Asked Questions (FAQ)”, USENET: comp.ai.genetic <news:comp.ai.genetic>. Available via anonymous FTP from rtfm.mit.edu/pub/usenet/news.answers/ai-faq/genetic/. About 110 pages.

[22] Ph. Kampas. “ Myoelektroden - optimal eingesetzt”, *Medizinisch-Orthopädische-Technik 1/2001*; 121. Jahrgang; Januar/Februar 2001; pp. 21-27, Gentner Verlag Stuttgart. English translation available at http://www.ottobock.se/info_download/pdf/Elektroden_Text_GB.pdf.

[23] J. Tørresen. “Two-Step Incremental Evolution of a Prosthetic Hand Controller Based on Digital Logic Gates”, *4th International Conference on Evolvable Hardware* (ICES2001), October 2001, Tokyo, Japan.

[24] Artificial Limb Services within Australia. “National indicator of prosthesis cost”, http://rehabtech.eng.monash.edu.au/techguide/als/als_1.idc.

[25] Motion Control Inc. “Pricing guidelines for a Utah artificial arm or Procontrol system”, http://www.utaharm.com/pdf/priceguide.pdf.

[26] I. Kajitani, I. Sekita, N. Otsu and T. Higuchi. “Improvements to the Action Decision Rate for a Multi-Function Prosthetic Hand”, *The First International Symposium on Measurement, Analysis and Modeling of Human Functions* (Proc. of ISHF2001), pp. 84-89, 2001.

[27] Motion Control Inc. “The Utah Procontrol 2 for below elbow or hybrid prostheses”, http://www.utaharm.com/procontrol.htm.

[28] Otto Bock. “13E195 Otto Bock Four Channel Processor II”, http://www.ottobockus.com/products/upper_limb_prosthetics/myoelectric_hands_10s17.asp.

[29] Otto Bock. “Otto Bock SensorHand”, http://www.ottobockus.com/products/upper_limb_prosthetics/myoelectric_hands_sensorhand.asp.

[30] J. Tørresen and V. E. Skaugen. “A Signal Processing Architecture Based On RAM Technology”, *Proceedings of 16th European Simulation Multiconference* (ESM-2002), pp. 317-319, Darmstadt, Germany, June 2002.